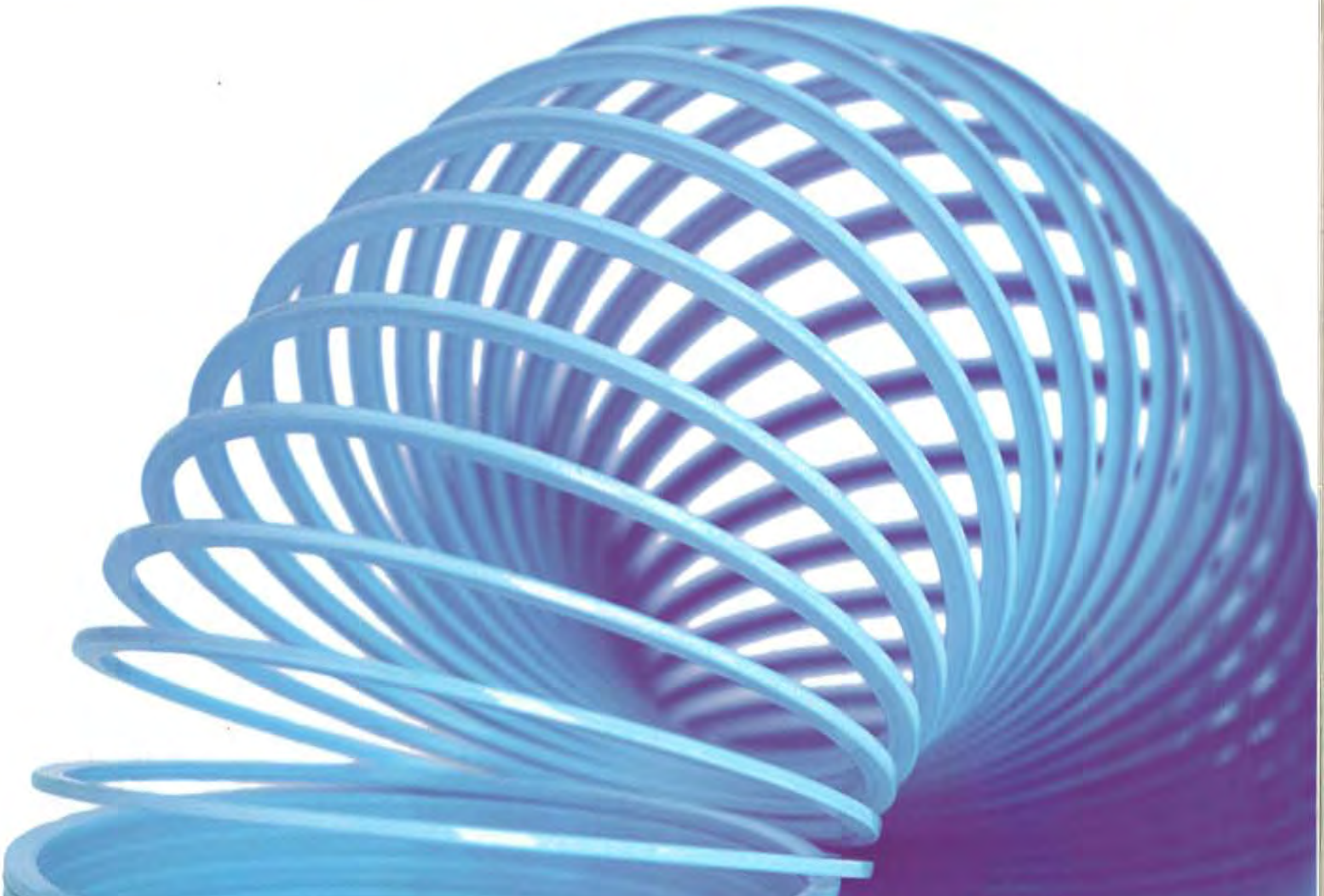
VISIT…

LANZAROTE
Caliente.com

STARTING OUT WITH PYTHON®



TONY GADDIS

STARTING OUT WITH

Python®

Tony Gaddis
Haywood Community College

PEARSON
Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Executive Editor	Michael Hirsch
Editorial Assistant	Stephanie Sellinger
Associate Managing Editor	Jeffrey Holcomb
Text Designer	Joyce Cosentino Wells
Cover Designer	Beth Paquin
Photo Research	Beth Anderson
Digital Assets Manager	Marianne Groth
Senior Media Producer	Bethany Tidd
Marketing Manager	Erin Davis
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Senior Media Buyer	Ginny Michaud
Production Coordination	Shelley Creager, Aptara Corp.
Composition and Illustrations	Aptara Corp.
Indexing	Steve Rath

Photo Credits
Cover image © Getty Images / Image Source Pink
Figure 1-3, "The ENIAC computer," (page 4) is courtesy of U.S. Army Historic Computer Images.
Figure 1-4, "A lab technician holds a modern microprocessor," (page 4) is courtesy of Intel Corporation.
Figure 1-5, "Memory chips," (page 5) is courtesy of IBM Corporation.
Rendered art and photographic images in Figures 1-2 (page 3), 1-15 (page 12), 1-16 and 1-17 (page 15), and 1-19 and 1-20 (page 19) © 2007 JUPITERIMAGES and its licensors. All Rights Reserved.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Gaddis, Tony.
Starting out with Python / Tony Gaddis.
p. cm.
Includes index.
ISBN-13: 978-0-321-53711-9
ISBN-10: 0-321-53711-4
1. Python (Computer program language) I. Title.

QA76.73.P98G34 2009
005.13'3--dc22
2008001684

Copyright © 2009 Pearson Education, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, fax (617) 671-3447, or online at http://www.pearsoned.com/legal/permissions.htm.

ISBN-13: 978-0-321-53711-9
ISBN-10: 0-321-53711-4

1 2 3 4 5 6 7 8 9 10—EB—12 11 10 09 08

Contents at a Glance

Preface xi

Part I: Programming Fundamentals

Chapter 1 Introduction to Computers and Programming 1
Chapter 2 Input, Processing, and Output 31
Chapter 3 Simple Functions 77
Chapter 4 Decision Structures and Boolean Logic 113
Chapter 5 Repetition Structures 151
Chapter 6 Value-Returning Functions and Modules 191

Part II: Using Objects to Perform Tasks

Chapter 7 Files and Exceptions 225
Chapter 8 Working with Sequences: Strings and Lists 273

Part III: Object-Oriented Programming

Chapter 9 Classes and Object-Oriented Programming 329
Chapter 10 Inheritance 373

Part IV: Advanced Topics

Chapter 11 Recursion 399
Chapter 12 GUI Programming 419

Appendix A Installing Python 457
Appendix B Introduction to IDLE 459
Appendix C The ASCII Character Set 467
Index 469

Student CD The following appendix is on the accompanying Student CD.

Appendix D Answers to Checkpoints

Contents

Preface xi

Part I: **Programming Fundamentals**

Chapter 1 **Introduction to Computers and Programming** 1
1.1 Introduction 1
1.2 Hardware and Software 2
1.3 How Computers Store Data 8
1.4 How a Program Works 13
1.5 Using Python 20

Chapter 2 **Input, Processing, and Output** 31
2.1 Designing a Program 31
2.2 Input, Processing, and Output 35
2.3 Displaying Output with the `print` Statement 36
2.4 Comments 39
2.5 Variables 40
2.6 Reading Input from the Keyboard 48
2.7 Performing Calculations 50
2.8 More About Data Output 62

Chapter 3 **Simple Functions** 77
3.1 Introduction to Functions 77
3.2 Defining and Calling a Function 79
3.3 Designing a Program to Use Functions 85
3.4 Local Variables 91
3.5 Passing Arguments to Functions 93
3.6 Global Variables and Global Constants 103

Chapter 4 **Decision Structures and Boolean Logic** 113
4.1 The `if` Statement 113
4.2 The `if-else` Statement 121
4.3 Comparing Strings 126
4.4 Nested Decision Structures and the `if-elif-else` Statement 130
4.5 Logical Operators 138
4.6 Boolean Variables 144

Chapter 5 **Repetition Structures** 151

5.1 Introduction to Repetition Structures 151
5.2 The `while` Loop: a Condition-Controlled Loop 152
5.3 The `for` Loop: a Count-Controlled Loop 161
5.4 Calculating a Running Total 173
5.5 Sentinels 176
5.6 Input Validation Loops 179
5.7 Nested Loops 184

Chapter 6 **Value-Returning Functions and Modules** 191

6.1 Introduction to Value-Returning Functions: Generating Random Numbers 191
6.2 Writing Your Own Functions 200
6.3 The `math` Module 211
6.4 Storing Functions in Modules 214

Part II: **Using Objects to Perform Tasks**

Chapter 7 **Files and Exceptions** 225

7.1 Introduction to File Input and Output 225
7.2 Using Loops to Process Files 242
7.3 Processing Records 249
7.4 Exceptions 262

Chapter 8 **Working with Sequences: Strings and Lists** 273

8.1 Sequences 273
8.2 Working with Strings 273
8.3 Lists 294

Part III: **Object-Oriented Programming**

Chapter 9 **Classes and Object-Oriented Programming** 329

9.1 Procedural and Object-Oriented Programming 329
9.2 Classes 333
9.3 Working with Instances 350
9.4 Techniques for Designing Classes 358

Chapter 10 **Inheritance** 373

10.1 Introduction to Inheritance 373
10.2 Polymorphism 388

Part IV: **Advanced Topics**

Chapter 11 **Recursion** 399

11.1 Introduction to Recursion 399
11.2 Problem Solving with Recursion 402
11.3 Examples of Recursive Algorithms 406

Chapter 12 **GUI Programming** 419

12.1 Graphical User Interfaces 419
12.2 Using the `Tkinter` Module 421
12.3 Display Text with `Label` Widgets 424
12.4 Organizing Widgets with Frames 427
12.5 `Button` Widgets and Info Dialog Boxes 430
12.6 Getting Input with the `Entry` Widget 433
12.7 Using Labels as Output Fields 436
12.8 Radio Buttons and Check Buttons 444

Appendix A **Installing Python** 457
Appendix B **introduction to IDLE** 459
Appendix C **The ASCII Character Set** 467
Index 469

Student CD The following appendix is on the accompanying Student CD.

Appendix D **Answers to Checkpoints**

Preface

Welcome to *Starting Out with Python*. This book uses the Python language to teach programming concepts and problem-solving skills, without assuming any previous programming experience. With easy-to-understand examples, pseudocode, flowcharts, and other tools, the student learns how to design the logic of programs and then implement those programs using Python. This book is ideal for an introductory programming course or a programming logic and design course using Python as the language.

As with all the books in the *Starting Out With* series, the hallmark of this text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in example programs that are concise and practical. The programs in this book include short examples that highlight specific programming topics, as well as more involved examples that focus on problem solving. Each chapter provides one or more case studies that provide step-by-step analysis of a specific problem and shows the student how to solve it.

Control Structures First, Then Classes

Python is a fully object-oriented programming language, but students do not have to understand object-oriented concepts to start programming in Python. This text first introduces the student to the fundamentals of data storage, input and output, control structures, functions, sequences and lists, file I/O, and objects that are created from standard library classes. Then the student learns to write classes, explores the topics of inheritance and polymorphism, and learns to write recursive functions. Finally, the student learns to develop simple event-driven GUI applications.

Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Programming

This chapter begins by giving a very concrete and easy-to-understand explanation of how computers work, how data is stored and manipulated, and why we write programs in high-level languages. An introduction to Python, interactive mode, script mode, and the IDLE environment is also given.

Chapter 2: Input, Processing, and Output

This chapter introduces the program development cycle, variables, data types, and simple programs that are written as sequence structures. The student learns to write simple programs that read input from the keyboard, perform mathematical operations, and produce screen output. Pseudocode and flowcharts are also introduced as tools for designing programs.

Chapter 3: Simple Functions

This chapter shows the benefits of modularizing programs and using the top-down design approach. The student learns to define and call simple functions (functions that do not return values), pass arguments to functions, and use local variables. Hierarchy charts are introduced as a design tool.

Chapter 4: Decision Structures and Boolean Logic

In this chapter the student learns about relational operators and Boolean expressions and is shown how to control the flow of a program with decision structures. The `if`, `if-else`, and `if-elif-else` statements are covered. Nested decision structures and logical operators are also discussed.

Chapter 5: Repetition Structures

This chapter shows the student how to create repetition structures using the `while` loop and `for` loop. Counters, accumulators, running totals, and sentinels are discussed, as well as techniques for writing input validation loops.

Chapter 6: Value-Returning Functions and Modules

This chapter begins by discussing common library functions, such as those for generating random numbers. After learning how to call library functions and use their return value, the student learns to define and call his or her own functions. Then the student learns how to use modules to organize functions.

Chapter 7: Files and Exceptions

This chapter introduces sequential file input and output. The student learns to read and write large sets of data and store data as fields and records. The chapter concludes by discussing exceptions and shows the student how to write exception-handling code.

Chapter 8: Working with Sequences: Strings and Lists

This chapter introduces the student to the concept of a sequence in Python and explores the use of two common Python sequences: strings and lists. Several programming techniques are shown using strings with operators, built-in functions, library functions, and string methods. The student also learns to use lists for array-like processing.

Chapter 9: Classes and Object-Oriented Programming

This chapter compares procedural and object-oriented programming practices. It covers the fundamental concepts of classes and objects. Attributes, methods, encapsulation and data hiding, `__init__` functions (which are similar to constructors), accessors, and mutators are discussed. The student learns how to model classes with UML and how to find the classes in a particular problem.

Chapter 10: Inheritance

The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include superclasses, subclasses, how `__init__` functions work in inheritance, method overriding, and polymorphism.

Chapter 11: Recursion

This chapter discusses recursion and its use in problem solving. A visual trace of recursive calls is provided and recursive applications are discussed. Recursive algorithms for many tasks are presented, such as finding factorials, finding a greatest common denominator (GCD), and summing a range of values in a list, and the classic Towers of Hanoi example are presented.

Chapter 12: GUI Programming

This chapter discusses the basic aspects of designing a GUI application using the *tkinter* module in Python. Fundamental widgets, such as labels, button, entry fields, radio buttons, check buttons, and dialog boxes, are covered. The student also learns how events work in a GUI application and how to write callback functions to handle events.

Appendix A: Installing Python

This appendix explains how to install the Python interpreter from the accompanying CD or download it from the Python Web site.

Appendix B: Introduction to IDLE

This appendix gives an overview of the IDLE integrated development environment that comes with Python.

Appendix C: The ASCII Character Set

As a reference, this appendix lists the ASCII character set.

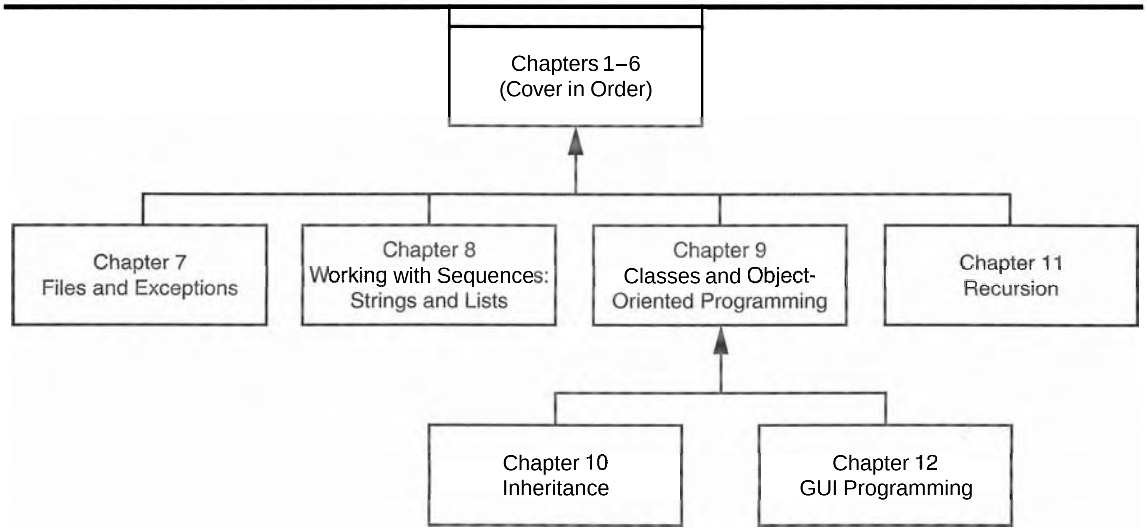
Appendix D: Answers to Checkpoint Questions

This appendix gives the answers to the Checkpoint questions that appear throughout the text.

Organization of the Text

The text teaches programming in a step-by-step manner. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, you do have some flexibility in the order that you wish to cover them. Figure P-1 shows chapter dependencies. Each box represents a chapter or a group of chapters. An arrow points from a chapter to the chapter that must be covered before it.

Figure P-1 Chapter dependencies


Chapters 1–6
(Cover in Order)
Chapter 7
Files and Exceptions
Chapter 8
Working with Sequences:
Strings and Lists
Chapter 9
Classes and Object-
Oriented Programming
Chapter 11
Recursion
Chapter 10
Inheritance
Chapter 12
GUI Programming

Features of the Text

Concept Statements	Each major section of the text starts with a concept statement. This statement concisely summarizes the main point of the section.
Example Programs	Each chapter has an abundant number of complete and partial example programs, each designed to highlight the current topic.
In the Spotlight Case Studies	Each chapter has one or more In *the* Spotlight case studies that provide detailed, step-by-step analysis of problems and show the student how to solve them.
Notes	Notes appear at several places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.
Tips	Tips advise the student on the best techniques for approaching different programming problems.
Warnings	Warnings caution students about programming techniques or practices that can lead to malfunctioning programs or lost data.
Checkpoints	Checkpoints are questions placed at intervals throughout each chapter. They are designed to query the student's knowledge quickly after learning a new topic.
Review Questions	Each chapter presents a thorough and diverse set of review questions and exercises. They include Multiple Choice, True/False, Algorithm Workbench, and Short Answer.
Programming Exercises	Each chapter offers a pool of programming exercises designed to solidify the student's knowledge of the topics currently being studied.

Supplements

Student Resource CD

This CD includes:

- The Python Interpreter, including the IDLE programming environment
- All of the book's example programs
- Appendix D: Answers to Checkpoint Questions

If a CD did not come with your book or you can't locate your CD, visit `http://www.aw.com/cssupport/` to access most of these items.

Instructor Resources

The following supplements are available to qualified instructors only:

- Answers to all of the Review Questions
- Solutions for the exercises
- PowerPoint presentation slides for each chapter
- Test bank

Visit the Addison-Wesley Instructor Resource Center (`www.aw.com/irc`) or send an email to `computing@aw.com` for information on how to access them.

Acknowledgments

I want to thank everyone at Addison-Wesley for making the *Starting Out With* series so successful. I am extremely grateful to Michael Hirsch, executive editor, and Stephanie Sellinger, editorial assistant, for guiding me through the process of writing this book. I also want to thank Erin Davis for all of her work as marketing manager. I had a great production team for this book, led by Jeff Holcomb and including Shelley Creager, Brian Baker, David Lindsay (copyeditor), Joyce Cosentino Wells (text design), Beth Paquin (cover design), Bethany Tidd (media), Carol Melville (manufacturing), and Marianne Groth (supplements). Thanks to you all!

Last, but not least, I want to thank my family for all the patience, love, and support they have shown me throughout this and my many other projects.

About the Author

Tony Gaddis is the principal author of the *Starting Out With* series of textbooks. Tony has nearly two decades of experience teaching computer science courses, primarily at Haywood Community College. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College "Teacher of the Year" and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out With* series includes introductory books covering C++, Java™, Microsoft® Visual Basic^{B}, Microsoft® C#®, Python^{B}, and Alice, all published by Addison-Wesley. More information about all these books can be found at `www.gaddisbooks.com`.

CHAPTER

1 Introduction to Computers and Programming

TOPICS

1.1 Introduction
1.2 Hardware and Software
1.3 How Computers Store Data
1.4 How a Program Works
1.5 Using Python

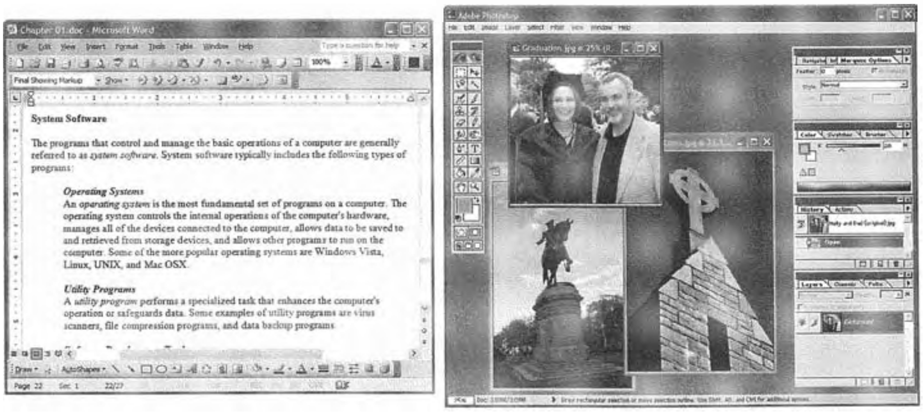
1.1 Introduction

Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and do many other things. At home, people use computers for tasks such as paying bills, shopping online, communicating with friends and family, and playing computer games. And don't forget that cell phones, iPods®, BlackBerries®, car navigation systems, and many other devices are computers too, The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed. This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. A program is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens from two commonly used programs, Microsoft Word and Adobe Photoshop. Microsoft Word is a word processing program that allows you to create, edit, and print documents with your computer. Adobe Photoshop is an image editing program that allows you to work with graphic images, such as photos taken with your digital camera.

Programs are commonly referred to as software. Software is essential to a computer because it controls everything the computer does. All of the software that we use to make our computers useful is created by individuals working as programmers or software developers. A programmer, or software developer, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers' work used in business, medicine, government, law enforcement, agriculture, academics, entertainment, and many other fields.

Figure 1-1 A word processing program and an image editing program



System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

Operating Systems
An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Some of the more popular operating systems are Windows Vista, Linux, UNIX, and Mac OSX.

Utility Programs
A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Some examples of utility programs are virus scanners, file compression programs, and data backup programs.

This book introduces you to the fundamental concepts of computer programming using the Python language. Before we begin exploring those concepts, you need to understand a few basic things about computers and how they work. This chapter will build a solid foundation of knowledge that you will continually rely on as you study computer science. First, we will discuss the physical components that computers are commonly made of. Next, we will look at how computers store data and execute programs. Finally, we will get a quick introduction to the software that you will use to write Python programs.

1.2 Hardware and Software

CONCEPT: The physical devices that a computer is made of are referred to as the computer's hardware. The programs that run on a computer are referred to as software.

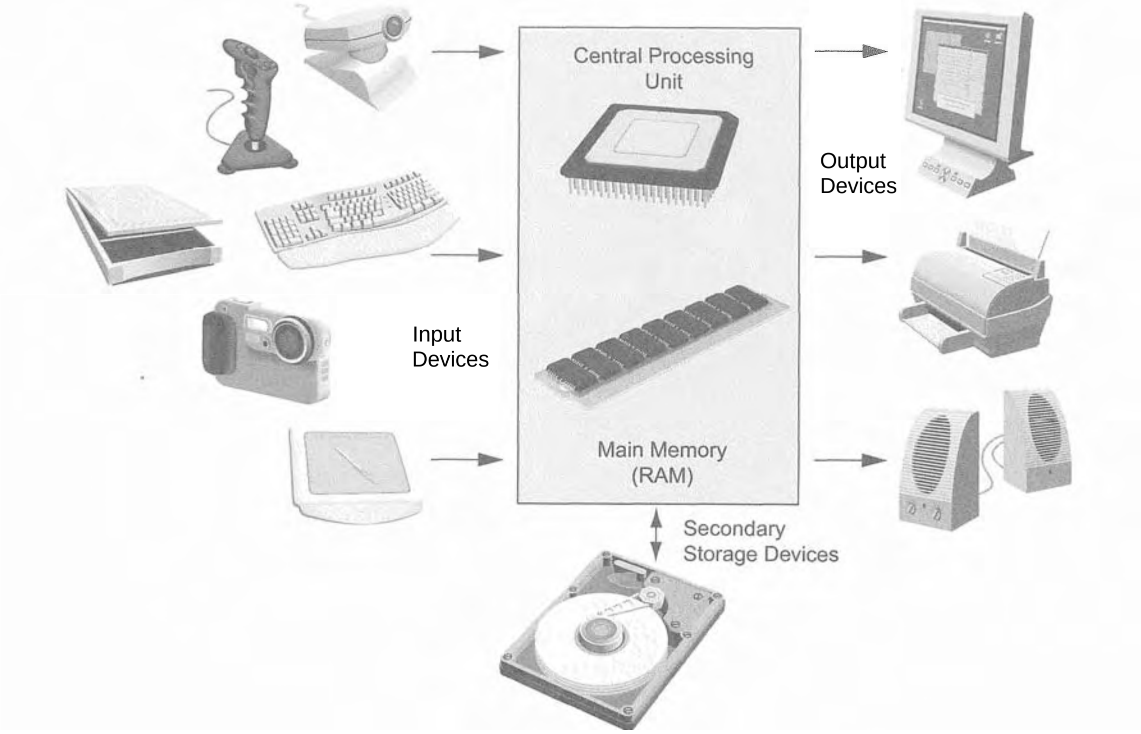
Hardware

The term *hardware* refers to all of the physical devices, or *components*, that a computer is made of. A computer is not one single device, but a system of devices that all work together. Like the different instruments in a symphony orchestra, each device in a computer plays its own part.

If you have ever shopped for a computer, you've probably seen sales literature listing components such as microprocessors, memory, disk drives, video displays, graphics cards, and so on. Unless you already know a lot about computers, or at least have a friend that does, understanding what these different components do might be challenging. As shown in Figure 1-2, a typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

Figure 1-2 Typical components of a computer system



Let's take a closer look at each of these components.

The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is *running* or *executing* the program. The *central processing unit,* or CPU, is the part of a computer that actually runs programs. The CPU is the most important component in a computer because without it, the computer could not run software.

In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two women in the photo are working with the historic ENIAC computer. The ENIAC, which is considered by many to be the world's first programmable electronic computer, was built in 1345 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall, 100 feet long, and weighed 30 tons.

Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern microprocessor. In addition to being much smaller than the old electromechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-3 The ENIAC computer (courtesy of U.S. Army Historic Computer Images)

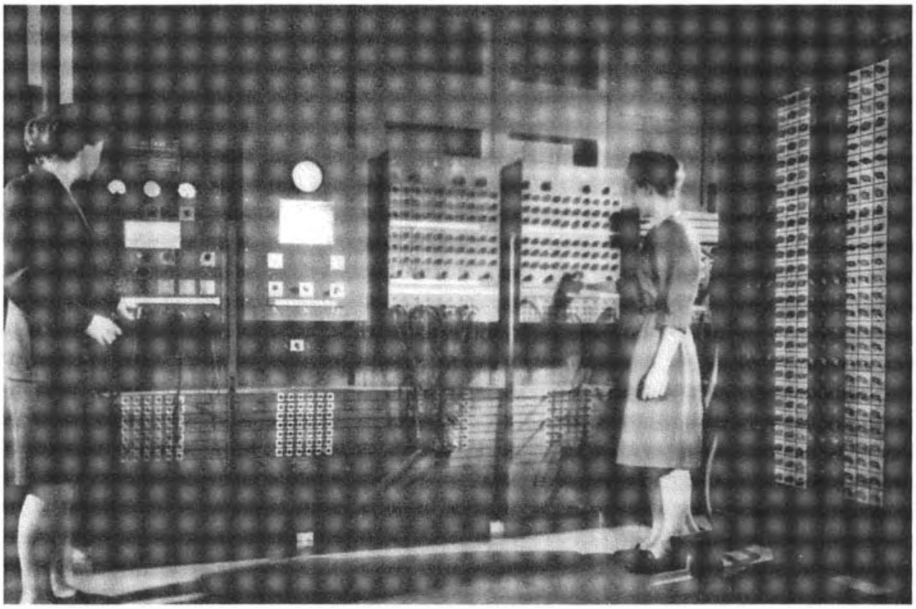


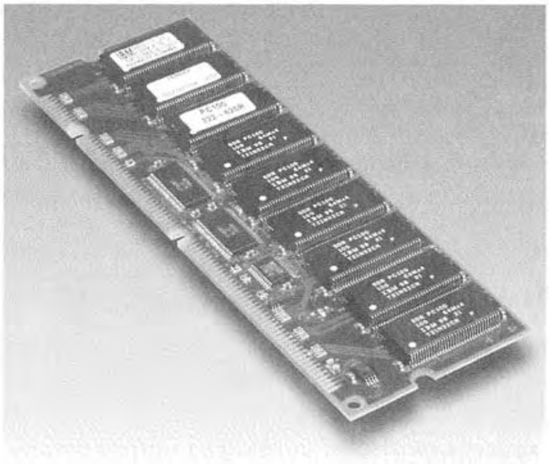
Figure 1-4 A lab technician holds a modern microprocessor (photo courtesy of Intel Corporation)

Main Memory

You can think of main memory as the computer's work area. This is where the computer stores a program while the program is running, as well as the data that the program is working with. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

Main memory is commonly known as *random-access memory,* or *RAM.* It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a *volatile* type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in chips, similar to the ones shown in Figure 1-5.

Figure 1-5 Memory chips (photo courtesy of IBM Corporation)



Secondary Storage Devices

Secondary storage is a type of memory that can hold data for long periods of time, even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory records, is saved to secondary storage as well.

The most common type of secondary storage device is the disk drive. A *disk drive* stores data by magnetically encoding it onto a circular disk. Most computers have a disk drive mounted inside their case. External disk drives, which connect to one of the computer's communication ports, are also available. External disk drives can be used to create backup copies of important data or to move data to another computer.

In addition to external disk drives, many types of devices have been created for copying data, and for moving it to other computers. For many years floppy disk drives were popular. A *floppy disk drive* records data onto a small floppy disk, which can be removed from the drive. Floppy disks have many disadvantages, however. They hold only a small amount of data, are slow to access data, and can be unreliable. The use of floppy disk drives has declined dramatically in recent years, in favor of superior devices such as USB drives. *USB drives* are small devices that plug into the computer's USB (universal serial bus) port, and appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives, which are also known as *memory sticks* and *flash drives,* are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the CD (compact disc) and the DVD (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they are good mediums for creating backup copies of data.

Input Devices

Input is any data the computer collects from people and from other devices. The component that collects the data and sends it to the computer is called an *input device.* Common input devices are the keyboard, mouse, scanner, microphone, and digital camera. Disk drives and optical drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer produces for people or for other devices. It might be a sales report, a list of names, or a graphic image. The data is sent to an *output device,* which formats and presents it. Common output devices are video displays and printers. Disk drives and CD recorders can also be considered output devices because the system sends data to them in order to be saved.

Software

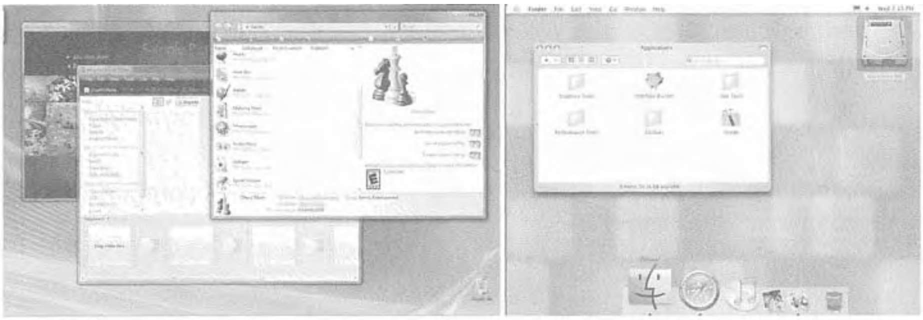
If a computer is to function, software is not optional. Everything that a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software.* System software typically includes the following types of programs:

Operating Systems An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Figure 1-6 shows screens from three popular operating systems: Windows Vista, Mac OS X, and Linux.

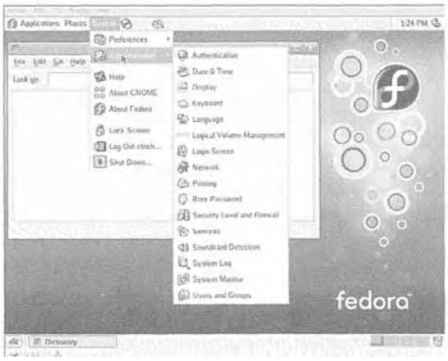
Figure 1-6 Screens From the Windows Vista, Mac OS X, and Fedora Linux operating systems



Windows Vista

Mac OS X

Fedora Linux



Utility Programs A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file compression programs, and data backup programs.

Software Development Tools Software development tools are the programs that programmers use to create, modify, and test software. Assemblers, compilers, and interpreters are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as *application software*. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two commonly used applications: Microsoft Word, a word processing program, and Adobe Photoshop, an image editing program. Some other examples of application software are spreadsheet programs, email programs, web browsers, and game programs.

Checkpoint

1.1 What is a program?

1.2 What is hardware?

1.3 List the five major components of a computer system.

1.4 What part of the computer actually runs programs?

1.5 What part of the computer serves as a work area to store a program and its data while the program is running?

1.6 What part of the computer holds data for long periods of time, even when there is no power to the computer?

1.7 What part of the computer collects data from people and from other devices?

1.8 What part of the computer formats and presents data for people or other devices?

1.9 What fundamental set of programs control the internal operations of the computer's hardware?

1.10 What do you call a program that performs a specialized task, such as a virus scanner, a file compression program, or a data backup program?

1.11 Word processing programs, spreadsheet programs, email programs, web browsers, and game programs belong to what category of software?

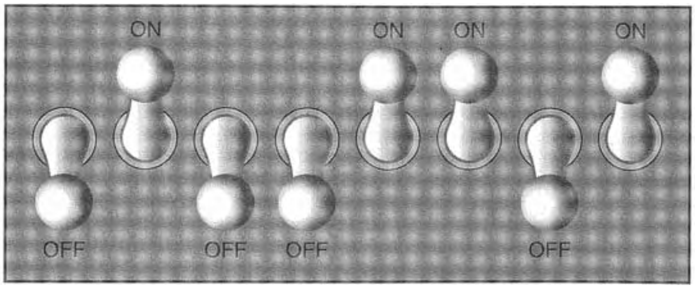
1.3 How Computers Store Data

CONCEPT: All data that is stored in a computer is converted to sequences of 0s and 1s.

A computer's memory is divided into tiny storage locations known as *bytes*. One byte is only enough memory to store a letter of the alphabet or a small number. In order to do anything meaningful, a computer has to have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

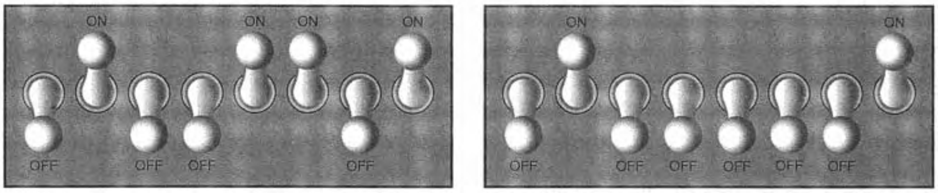
Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit.* Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position, and a negative charge as a switch in the *off* position. Figure 1-7 shows the way that a computer scientist might think of a byte of memory: as a collection of switches that are each flipped to either the on or off position.

Figure 1-7 Think of a byte as eight switches



When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data. For example, the pattern shown on the left in Figure 1-8 shows how the number 77 would be stored in a byte, and the pattern on the right shows how the letter A would be stored in a byte. We explain below how these patterns are determined.

Figure 1-8 Bit patterns for the number 77 and the letter A



The number 77 stored in a byte.

The letter A stored in a byte.

Storing Numbers

A bit can be used in a very limited way to represent numbers. Depending on whether the bit is turned on or off, it can represent one of two different values. In computer systems, a bit that is turned off represents the number 0 and a bit that is turned on represents the number 1. This corresponds perfectly to the binary numbering system. In the binary numbering system (or binary, as it is usually called) all numeric values are written as sequences of 0s and 1s. Here is an example of a number that is written in binary:

```
10011101
```

The position of each digit in a binary number has a value assigned to it. Starting with the rightmost digit and moving left, the position values are 2^0, 2^1, 2^2, 2^3, and so forth, as shown in Figure 1-9. Figure 1-10 shows the same diagram with the position values calculated. Starting with the rightmost digit and moving left, the position values are 1, 2, 4, 8, and so forth.

Figure 1-9 The values of binary digits as powers of 2

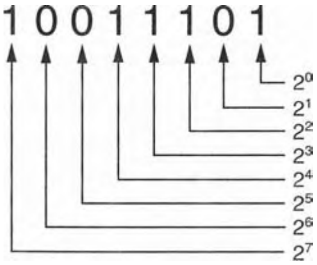
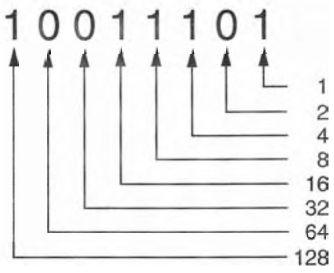


Figure 1-10 The values of binary digits


1 0 0 1 1 1 0 1
1
2
4
8
16
32
64
128

To determine the value of a binary number you simply add up the position values of all the 1s. For example, in the binary number 10011101, the position values of the 1s are 1, 4, 8, 16, and 128. This is shown in Figure 1-11. The sum of all of these position values is 157. So, the value of the binary number 10011101 is 157.

Figure 1-11 Determining the value of 10011101

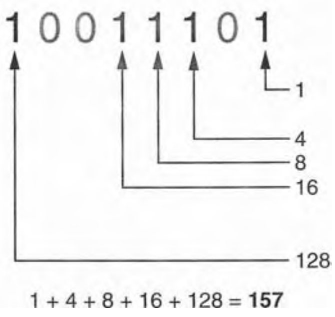
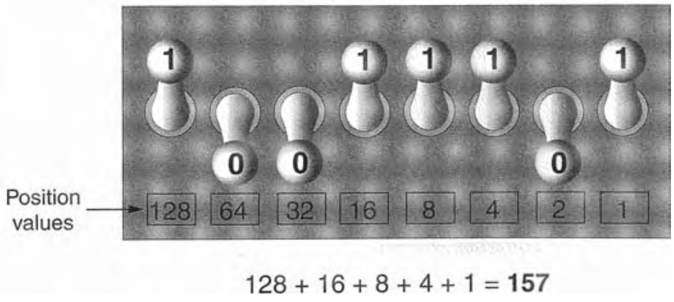

1 0 0 1 1 1 0 1
1
4
8
16
128
1 + 4 + 8 + 16 + 128 = 157

Figure 1-12 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

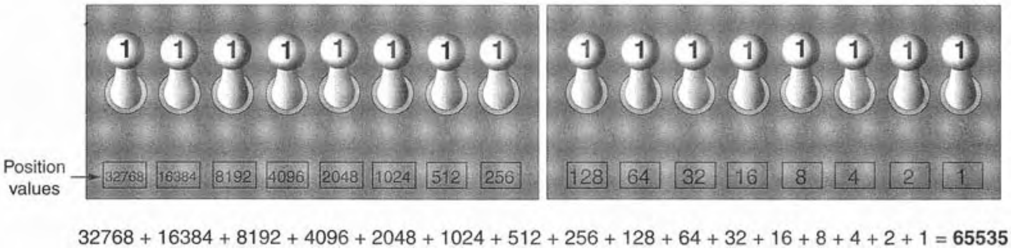
Figure 1-12 The bit pattern for 157


1
0
0
1
1
1
0
1
Position values
128
64
32
16
8
4
2
1
128 + 16 + 8 + 4 + 1 = 157

When all of the bits in a byte are set to 0 (turned off), then the value of the byte is 0. When all of the bits in a byte are set to 1 (turned on), then the byte holds the largest value that can be stored in it. The largest value that can be stored in a byte is 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. This limit exists because there are only eight bits in a byte.

What if you need to store a number larger than 255? The answer is simple: use more than one byte. For example, suppose we put two bytes together. That gives us 16 bits. The position values of those 16 bits would be 2^0, 2^1, 2^2, 2^3, and so forth, up through 2^{15}. As shown in Figure 1-13, the maximum value that can be stored in two bytes is 65,535. If you need to store a number iarger than this, then more bytes are necessary.

Figure 1-13 Two bytes used for a large number



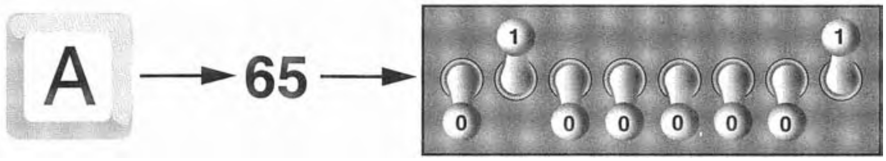
> TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

Storing Characters

Any piece of data that is stored in a computer's memory must be stored as a binary number. That includes characters, such as letters and punctuation marks. When a character is stored in memory, it is first converted to a numeric code. The numeric code is then stored in memory as a binary number.

Over the years, different coding schemes have been developed to represent characters in computer memory. Historically, the most important of these coding schemes is *ASCII*, which stands for the *American Standard Code for Information Interchange.* ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters. For example, the ASCII code for the uppercase letter A is 65. When you type an uppercase A on your computer keyboard, the number 65 is stored in memory (as a binary number, of course). This is shown in Figure 1-14.

Figure 1-14 The letter A is stored in memory as the number *65*



TIP: The acronym ASCII is pronounced "askee."

In case you are curious, the ASCII code for uppercase B is 66, for uppercase C is 67, and so forth. Appendix C shows all of the ASCII codes and the characters they represent.

The ASCII character set was developed in the early 1960s, and was eventually adopted by most all computer manufacturers. ASCII is limited however, because it defines codes for only 128 characters. To remedy this, the Unicode character set was developed in the early 1990s. *Unicode* is an extensive encoding scheme that is compatible with ASCII, but can also represent characters for many of the languages in the world. Today, Unicode is quickly becoming the standard character set used in the computer industry.

Advanced Number Storage

Earlier you read about numbers and how they are stored in memory. While reading that section, perhaps it occurred to you that the binary numbering system can be used to represent only integer numbers, beginning with 0. Negative numbers and real numbers (such as 3.14159) cannot be represented using the simple binary numbering technique we discussed.

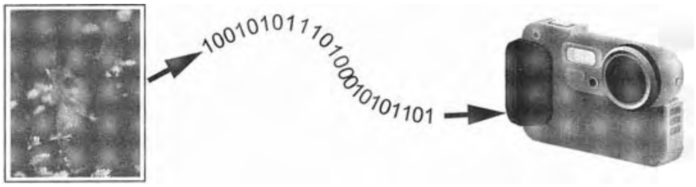
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as *two's complement,* and real numbers are encoded in *floating-point notation.* You don't need to know how these encoding schemes work, only that they are used to convert negative numbers and real numbers to binary format.

Other Types of Data

Computers are often referred to as digital devices. The term *digital* can be used to describe anything that uses binary numbers. *Digital data* is data that is stored in binary, and a *digital device* is any device that works with binary data. In this section we have discussed how numbers and characters are stored in binary, but computers also work with many other types of digital data.

For example, consider the pictures that you take with your digital camera. These images are composed of tiny dots of color known as *pixels.* (The term pixel stands for *picture element.)* As shown in Figure 1-15, each pixel in an image is converted to a numeric code that represents the pixel's color. The numeric code is stored in memory as a binary number.

Figure 1-15 A digital image is stored in binary format



The music that you play on your CD player, iPod or MP3 player is also digital. A digital song is broken into small pieces known as samples. Each sample is converted to a binary number, which can be stored in memory. The more samples that a song is divided into, the more it sounds like the original music when it is played back. A CD quality song is divided into more than 44,000 samples per second!

Checkpoint

1.12 What amount of memory is enough to store a letter of the alphabet or a small number?

1.13 What do you call a tiny "switch" that can be set to either on or off?

1.14 In what numbering system are all numeric values written as sequences of 0s and 1s?

1.15 What is the purpose of ASCII?

1.16 What encoding scheme is extensive enough to represent the characters of many of the languages in the world?

1.17 What do the terms "digital data" and "digital device" mean?

1.4 How a Program Works

CONCEPT: **A** computer's CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the "computer's brain," and is described as being "smart." Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value

As you can see from this list, the CPU performs simple operations on pieces of data. The CPU does nothing on its own, however. It has to be told what to do, and that's the purpose of a program. A program is nothing more than a list of instructions that cause the CPU to perform operations.

Each instruction in a program is a command that tells the CPU to perform a specific operation. Here's an example of an instruction that might appear in a program:

```
10110000
```

To you and me, this is only a series of 0s and 1s. To a CPU, however, this is an instruction to perform an operation.[1] It is written in 0s and 1s because CPUs only understand instructions that are written in *machine language,* and machine language instructions always have an underlying binary structure.

A machine language instruction exists for each operation that a CPU is capable of performing. For example, there is an instruction for adding numbers, there is an instruction for subtracting one number from another, and so forth. The entire set of instructions that a CPU can execute is known as the CPU's *instruction set.*

NOTE: There are several microprocessor companies today that manufacture CPUs. Some of the more well-known microprocessor companies are Intel, AMD, and Motorola. If you look carefully at your computer, you might find a tag showing a logo for its microprocessor.

Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

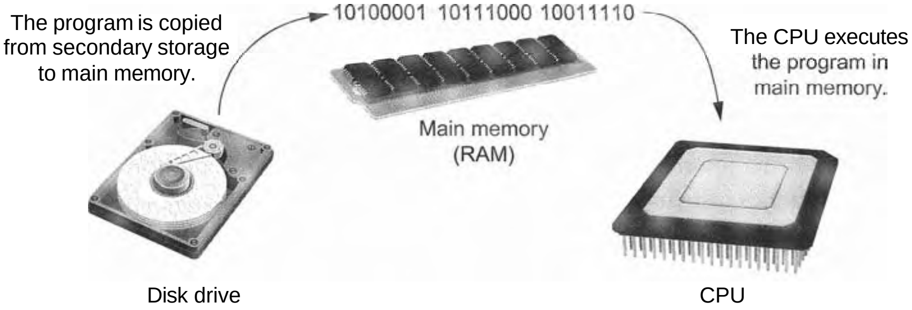
The machine language instruction that was previously shown is an example of only one instruction. It takes a lot more than one instruction, however, for the computer to do anything meaningful. Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations. For example, if you want your computer to calculate the amount of interest that you will earn from your savings account this year, the CPU will have to perform a large number of instructions, carried out in the proper sequence. It is not unusual for a program to contain thousands or even millions of machine language instructions.

Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically copied to your computer's disk drive from a CD-ROM, or perhaps downloaded from a website.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it. For example, suppose you have a word processing program on your computer's disk. To execute the program you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory. Then, the computer's CPU executes the copy of the program that is in main memory. This process is illustrated in Figure 1-16.

[1] The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

Figure 1-16 A program is copied into main memory and then executed

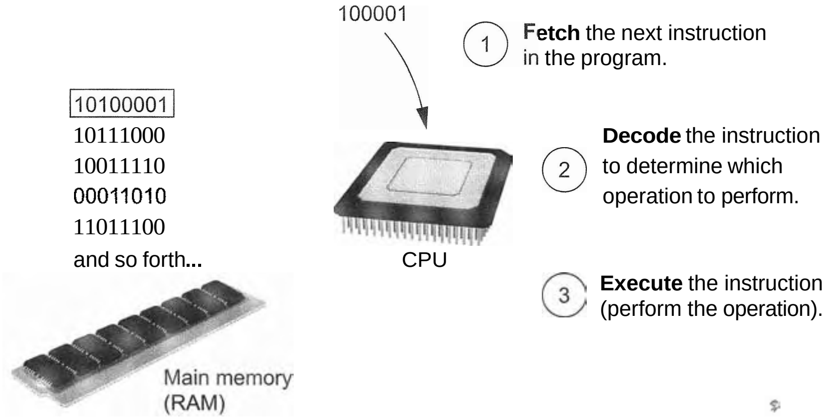

The program is copied from secondary storage to main memory.
10100001 10111000 10011110
The CPU executes the program in main memory.
Main memory (RAM)
Disk drive
CPU

When a CPU executes the instructions in a program, it is engaged in a process that is known as the *fetch-decode-execute cycle*. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are:

1. Fetch A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. Decode A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. Execute The last step in the cycle is to execute, or perform, the operation.

Figure 1-17 illustrates these steps.

Figure 1-17 The fetch-decode-execute cycle


100001
1
Fetch the next instruction in the program.
10100001
10111000
10011110
00011010
11011100
and so forth...
2
Decode the instruction to determine which operation to perform.
CPU
3
Execute the instruction (perform the operation).
Main memory (RAM)

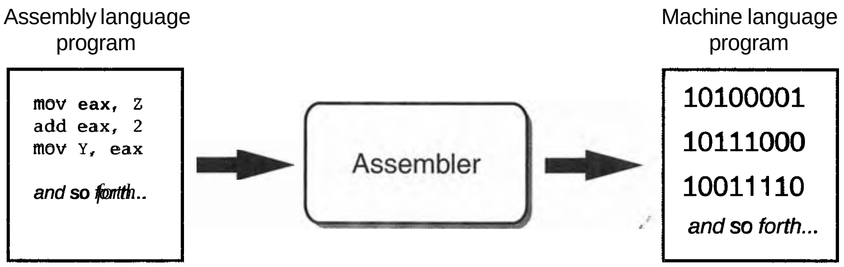
From Machine Language to Assembly Language

Computers can only execute programs that are written in machine language. As previously mentioned, a program can have thousands or even millions of binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult because putting a 0 or a 1 in the wrong place will cause an error, Although a computer's CPU only understands machine language, it is impractical for people to write programs in machine language. For this reason, *assembly language* was created in the early days of computing[2] as an alternative to machine language. Instead of using binary numbers for instructions, assembly language uses short words that are known as *mnemonics*. For example, in assembly language, the mnemonic `add` typically means to add numbers, `mul` typically means to multiply numbers, and mov typically means to move a value to a location in memory. When a programmer uses assembly language to write a program, he or she can write short mnemonics instead of binary numbers.

NOTE: There are many different versions of assembly language. It was mentioned earlier that each brand of CPU has its own machine language instruction set. Each brand of CPU typically has its own assembly language as well.

Assembly language programs cannot be executed by the CPU, however. The CPU only understands machine language, so a special program known as an *assembler* is used to translate an assembly language program to a machine language program. This process is shown in Figure 1-18. The machine language program that is created by the assembler can then be executed by the CPU.

Figure 1-18 An assembler translates an assembly language program to a machine language program



High-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a *low-level language*.

In the 1950s, a new generation of programming languages known as *high-level languages* began to appear. A high-level language allows you to create powerful and complex programs without knowing how the CPU works, and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), he or she would write the following instruction to display the message Hello world on the computer screen:

---

[2] The first assembly language was most likely that developed in the 1940s at Cambridge University for use with a historic computer known as the EDSAC.

```
DISPLAY "Hello world"
```

Python is a modern, high-level programming language that we will use in this book. In Python you would display the message Hello world with the following instruction:

```
print 'Hello world'
```

Doing the same thing in assembly language would require several instructions, and an intimate knowledge of how the CPU interacts with the computer's output device. As you can see from this example, high-level languages allow programmers to concentrate on the tasks they want to perform with their programs rather than the details of how the CPU will execute those programs.

Since the 1950s, thousands of high-level languages have been created. Table 1-1 lists several of the more well-known languages.

Table 1-1 Programming languages

Language	Description
Ada	Ada was created in the 1970s, primarily for applications used by the U.S. Department of Defense. The language is named in honor of Countess Ada Lovelace, an influential and historic figure in the field of computing.
BASIC	Beginners All-purpose Symbolic Instruction Code is a general-purpose language that was originally designed in the early 1960s to be simple enough for beginners to learn. Today, there are many different versions of BASIC.
FORTRAN	FORmula TRANslator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	Common Business-Oriented Language was created in the 1950s, and was designed for business applications.
Pascal	Pascal was created in 1970, and was originally designed for teaching programming. The language was named in honor of the mathematician, physicist, and philosopher Blaise Pascal.
C and C++	C and C++ (pronounced "c plus plus") are powerful, general-purpose languages developed at Bell Laboratories. The C language was created in 1972 and the C++ language was created in 1983.
C#	Pronounced "c sharp." This language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.
Java	Java was created by Sun Microsystems in the early 1990s. It can be used to develop programs that run on a single computer or over the Internet from a web server.
JavaScript	JavaScript, created in the 1990s, can be used in web pages. Despite its name, JavaScript is not related to Java.
Python	Python, the language we use in this book, is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on web servers.
Visual Basic	Visual Basic (commonly known as VB) is a Microsoft programming language and software development environment that allows programmers to create Windows-based applications quickly. VB was originally created in the early 1990s.

Key Words, Operators, and Syntax: an Overview

Each high-level language has its own set of predefined words that the programmer must use to write a program. The words that make up a high-level programming language are known as key words or reserved words. Each key word has a specific meaning, and cannot be used for any other purpose. You previously saw an example of a Python statement that uses the key word `print` to print a message on the screen. Table 1-2 shows all of the Python key words.

Table 1-2 The Python key words

and
as
assert
break
class
continue
def
del
elif
else
except
exec
finally
for
from
global
if
import
in
is
lambda
not
or
pass
print
raise
return
try
while
with
yiel

In addition to key words, programming languages have operators that perform various operations on data. For example, all programming languages have math operators that perform arithmetic. In Python, as well as most other languages, the + sign is an operator that adds two numbers. The following adds 12 and 75:

```
12 + 75
```

There are numerous other operators in the Python language, many of which you will learn about as you progress through this text.

In addition to key words and operators, each language also has its own syntax, which is a set of rules that must be strictly followed when writing a program. The syntax rules dictate how key words, operators, and various punctuation characters must be used in a program. When you are learning a programming language, you must learn the syntax rules for that particular language.

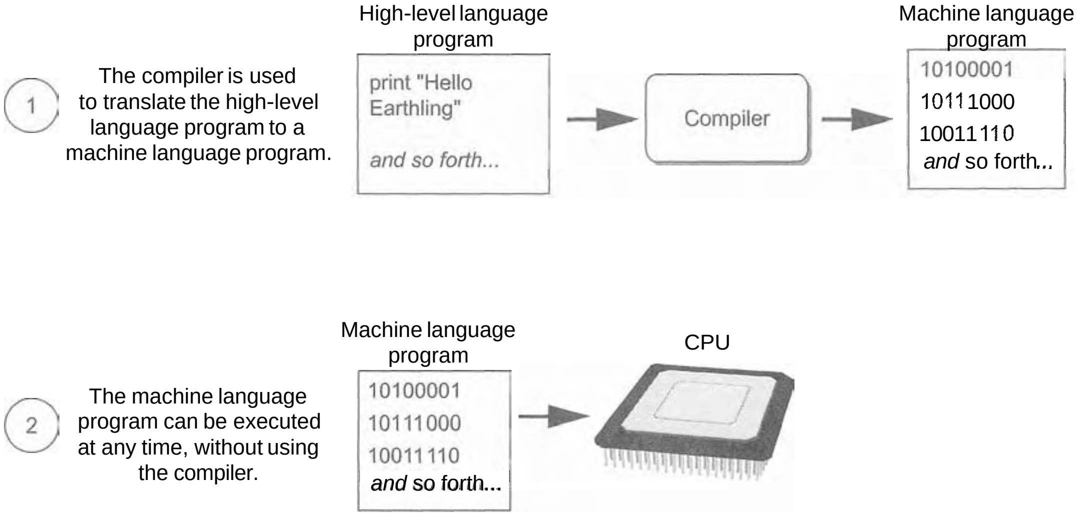
The individual instructions that you use to write a program in a high-level programming language are called statements. A programming statement can consist of key words, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.

Compilers and Interpreters

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Depending on the language that a program has been written in, the programmer will use either a compiler or an interpreter to make the translation.

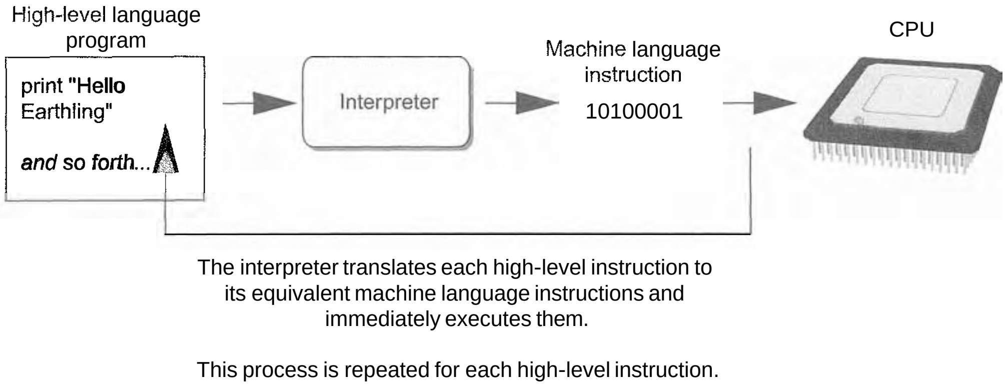
A *compiler* is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in Figure 1-19. As shown in the figure, compiling and executing are two different processes.

Figure 1-19 Compiling a high-level program and executing it


High-level language program
Machine language program
① The compiler is used to translate the high-level language program to a machine language program.
print "Hello Earthling"
and so forth...
Compiler
10100001
10111000
10011110
and so forth...
Machine language program
CPU
② The machine language program can be executed at any time, without using the compiler.
10100001
10111000
10011110
and so forth...

The Python language uses an interpreter, which is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to machine language instructions and then immediately executes them. This process repeats for every instruction in the program. This process is illustrated in Figure 1-20. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

Figure 1-28 Executing a high-level program with an interpreter


High-level language program
print "Hello Earthling"
and so forth...
Interpreter
Machine language instruction
10100001
CPU
The interpreter translates each high-level instruction to its equivalent machine language instructions and immediately executes them.
This process is repeated for each high-level instruction.

The statements that a programmer writes in a high-level language are called source code, or simply code. Typically, the programmer types a program's code into a text editor and then saves the code in a file on the computer's disk. Next, the programmer uses a compiler to translate the code into a machine language program, or an interpreter to translate and execute the code. If the code contains a syntax error, however, it cannot be translated. A ***syntax*** error is a mistake such as a misspelled key word, a missing punctuation character, or the incorrect use of an operator. When this happens the compiler or interpreter displays an error message indicating that the program contains a syntax error. The programmer corrects the error and then attempts once again to translate the program.

> **NOTE:** Human languages also have syntax rules. Do you remember when you took your first English class, and you learned all those rules about commas, apostrophes, capitalization, and so forth? You were learning the syntax of the English language.
>
> Although people commonly violate the syntax rules of their native language when speaking and writing, other people usually understand what they mean. Unfortunately, compilers and interpreters do not have this ability. If even a single syntax error appears in a program, the program cannot be compiled or executed. When an interpreter encounters a syntax error, it stops executing the program.

Checkpoint

1.18 A CPU understands instructions that are written only in what language?
1.19 A program has to be copied into what type of memory each time the CPU executes it?
1.20 When a CPU executes the instructions in a program, it is engaged in what process?
1.21 What is assembly language?
1.22 What type of programming language allows you to create powerful and complex programs without knowing how the CPU works?
1.23 Each language has a set of rules that must be strictly followed when writing a program. What is this set of rules called?
1.24 What do you call a program that translates a high-level language program into a separate machine language program?
1.25 What do you call a program that both translates and executes the instructions in a high-level language program?
1.26 What type of mistake is usually caused by a misspelled key word, a missing punctuation character, or the incorrect use of an operator?

1.5 Using Python

CONCEPT: The Python interpreter can run Python programs that are saved in files, or interactively execute Python statements that are typed at the keyboard. Python comes with a program named IDLE that simplifies the process of writing, executing, and testing programs.

Installing Python

Before you can try any of the programs shown in this book, or write any programs of your own, you need to make sure that Python is installed on your computer and properly configured. If you are working in a computer lab, this has probably been done already. If you are using your own computer, you can follow the instructions in Appendix A to install Python from the accompanying CD.

The Python Interpreter

You learned earlier that Python is an interpreted language. When you install the Python language on your computer, one of the items that is installed is the Python interpreter. The *Python interpreter* is a program that can read Python programming statements and execute them. (Sometimes we will refer to the Python interpreter simply as the interpreter.)

You can use the interpreter in two modes: interactive mode and script mode. In *interactive mode,* the interpreter waits for you to type Python statements on the keyboard. Once you type a statement, the interpreter executes it and then waits for you to type another statement. In *script mode,* the interpreter reads the contents of a file that contains Python statements. Such a file is known as a *Python program* or a *Python script.* The interpreter executes each statement in the Python program as it reads it.

Interactive Mode

Once Python has been installed and set up on your system, you start the interpreter in interactive mode by going to the operating system's command line and typing the following command:

```
python
```

If you are using Windows, you can alternatively click the *Start* button, then *All Programs*. You should see a program group named something like *Python 2.5.* (The *"2.5"* is the version of Python that is installed. At the time this is being written, Python 2.5 is the latest version.) Inside this program group you should see an item named *Python (command line)*. Clicking this menu item will start the Python interpreter in interactive mode.

When the Python interpreter starts in interactive mode, you will see something like the following displayed in a console window:

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` that you see is a prompt that indicates the interpreter is waiting for you to type a Python statement. Let's try it out. One of the simplest statements that you can write in Python is a `print` statement, which causes a message to be displayed on the screen. For example, the following statement causes the message `Python programming is fun!` to be displayed:

```
print 'Python programming is fun!'
```

Notice that after the word `print`, we have written `Python programming is fun!` inside a set of single-quote marks. The quote marks are necessary, but they will not be displayed. They simply mark the beginning and the end of the text that we wish to display. Here is an example of how you would type this `print` statement at the interpreter's prompt:

```
>>> print 'Python programming is fun!'
```

After typing the statement you press the Enter key and the Python interpreter executes the statement, as shown here:

```
>>> print 'Python programming is fun!' [ENTER]
Python programming is fun!
>>>
```

After the message is displayed, the >>> prompt appears again, indicating that the interpreter is waiting for you to enter another statement. Let's look at another example. In the following sample session we have entered two `print` statements.

```
>>> print 'To be or not to be' [ENTER]
To be or not to be
>>> print 'That is the question.' [ENTER]
That is the question.
>>>
```

If you incorrectly type a statement in interactive mode, the interpreter will display an error message. This will make interactive mode useful to you while you learn Python. As you learn new parts of the Python language, you can try them out in interactive mode and get immediate feedback from the interpreter.

To quit the Python interpreter in interactive mode on a Windows computer, press Ctrl-Z (pressing both keys together) followed by Enter. On a Mac, Linux, or UNIX computer, press Ctrl-D.

Writing Python Programs and Running Them in Script Mode

Although interactive mode is useful for testing code, the statements that you enter in interactive mode are not saved as a program. They are simply executed and their results displayed on the screen. If you want to save a set of Python statements as a program, you save those statements in a file. Then, to execute the program, you use the Python interpreter in script mode.

For example, suppose you want to write a Python program that displays the following three lines of text:

```
Nudge nudge
Wink wink
Know what I mean?
```

To write the program you would use a simple text editor like Notepad (which is installed on all Windows computers) to create a file containing the following statements:

```
print 'Nudge nudge'
print 'Wink wink'
print 'Know what I mean?'
```

NOTE: It is possible to use a word processor to create a Python program, but you must be sure to save the program as a plain text file. Otherwise the Python interpreter will not be able to read its contents.

When you save a Python program, you give it a name that ends with the `.py` extension, which identifies it as a Python program. For example, you might save the program previously shown with the name `test.py`. To run the program you would go to the directory in which the file is saved and type the following command at the operating system command line:

```
python test.py
```

This starts the Python interpreter in script mode and causes it to execute the statements in the file `test.py`. When the program finishes executing, the Python interpreter exits.

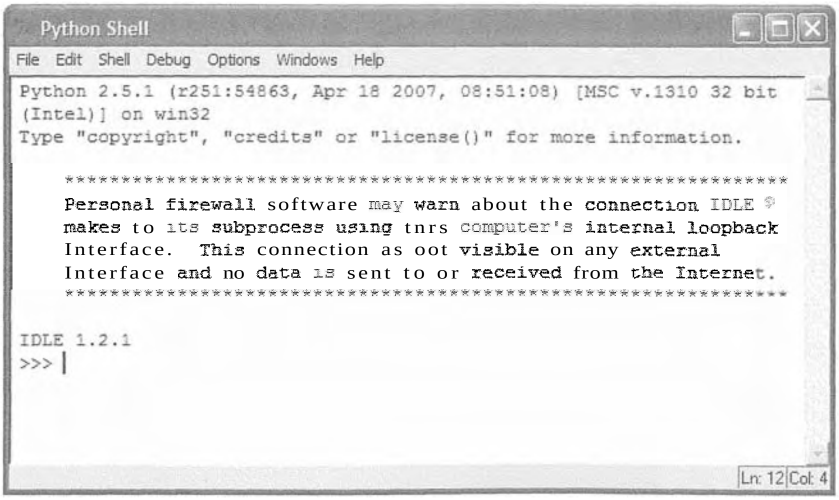
The IDLE Programming Environment

The previous sections described how the Python interpreter can be started in interactive mode or script mode at the operating system command line. As an alternative, you can use an *integrated development environment*, which is a single program that gives you all of the tools you need to write, execute, and test a program.

Recent versions of Python include a program named IDLE, which is automatically installed when the Python language is installed. (IDLE stands for Integrated Development Environment.) When you run IDLE, the window shown in Figure 1-21 appears. Notice that the >>> prompt appears in the IDLE window, indicating that the interpreter is running in interactive mode. You can type Python statements at this prompt and see them executed in the IDLE window.

IDLE also has a built-in text editor with features specifically designed to help you write Python programs. For example, the IDLE editor "colorizes" code so that key words and other parts of a program are displayed in their own distinct colors. This helps make programs easier to read. In IDLE you can write programs, save them to disk, and execute them. Appendix B provides a quick introduction to IDLE, and leads you through the process of creating, saving, and executing a Python program.

Figure **1-21** IDLE



```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using tnrs computer's internal loopback
    Interface.  This connection as oot visible on any external
    Interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> |
```

NOTE: Although IDLE is installed with Python, there are several other Python IDEs available. Your instructor might prefer that you use a specific one in class.

Review Questions

Multiple Choice

1. A(n) __________ is a set of instructions that a computer follows to perform a task.
    a. compiler
    b. program
    c. interpreter
    d. programming language
2. The physical devices that a computer is made of are referred to as __________.
    a. hardware
    b. software
    c. the operating system
    d. tools
3. The part of a computer that runs programs is called __________.
    a. RAM
    b. secondary storage
    c. main memory
    d. the CPU
4. Today, CPUs are small chips known as __________.
    a. ENIACs
    b. microprocessors
    c. memory chips
    d. operating systems
5. The computer stores a program while the program is running, as well as the data that the program is working with, in __________.
    a. secondary storage
    b. the CPU
    c. main memory
    d. the microprocessor
6. This is a volatile type of memory that is used only for temporary storage while a program is running.
    a. RAM
    b. secondary storage
    c. the disk drive
    d. the USB drive

7. A type of memory that can hold data for long periods of time, even when there is no power to the computer, is called _________.
   a. RAM
   b. main memory
   c. secondary storage
   d. CPU storage
8. A component that collects data from people or other devices and sends it to the computer is called _________.
   a. an output device
   b. an input device
   c. a secondary storage device
   d. main memory
9. A video display is a(n) _________ device.
   a. output device
   b. input device
   c. secondary storage device
   d. main memory
10. A _________ is enough memory to store a letter of the alphabet or a small number.
    a. byte
    b. bit
    c. switch
    d. transistor
11. A byte is made up of eight _________.
    a. CPUs
    b. instructions
    c. variables
    d. bits
12. In a(n) _________ numbering system, all numeric values are written as sequences of 0s and 1s.
    a. hexadecimal
    b. binary
    c. octal
    d. decimal
13. A bit that is turned off represents the following value: _________.
    a. 1
    b. −1
    c. 0
    d. "no"
14. A set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters is _________.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC

15. An extensive encoding scheme that can represent characters for many languages in the world is ________.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC
16. Negative numbers are encoded using the ________ technique.
    a. twos compliment
    b. floating point
    c. ASCII
    d. Unicode
17. Real numbers are encoded using the ________ technique.
    a. two's complement
    b. floating point
    c. ASCII
    d. Unicode
18. The tiny dots of color that digital images are composed of are called ________.
    a. bits
    b. bytes
    c. color packets
    d. pixels
19. If you were to look at a machine language program, you would see ________.
    a. Python code
    b. a stream of binary numbers
    c. English words
    d. circuits
20. In the ________ part of the fetch-decode-execute cycle, the CPU determines which operation it should perform.
    a. fetch
    b. decode
    c. execute
    d. immediately after the instruction is executed
21. Computers can only execute programs that are written in ________.
    a. Java
    b. assembly language
    c. machine language
    d. Python
22. The ________ translates an assembly language program to a machine language program.
    a. assembler
    b. compiler
    c. translator
    d. interpreter

23. The words that make up a high-level programming language are called _________.
    a. binary instructions
    b. mnemonics
    c. commands
    c. key words
24. The rules that must be followed when writing a program are called _________.
    a. syntax
    b. punctuation
    c. key words
    d. operators
25. A(n) _________ program translates a high-level language program into a separate machine language program.
    a. assembler
    b. compiler
    c. translator
    d. utility

True or False

1. Today, CPUs are huge devices made of electrical and mechanical components such as vacuum tubes and switches.
2. Main memory is also known as RAM.
3. Any piece of data that is stored in a computer's memory must be stored as a binary number.
4. Images, like the ones you make with your digital camera, cannot be stored as binary numbers.
5. Machine language is the only language that a CPU understands.
6. Assembly language is considered a high-level language.
7. An interpreter is a program that both translates and executes the instructions in a high-level language program.
8. A syntax error does not prevent a program from being compiled and executed.
9. Windows Vista, Linux, UNIX, and Mac OSX are all examples of application software.
10. Word processing programs, spreadsheet programs, email programs, web browsers, and games are all examples of utility programs.

Short Answer

1. Why is the CPU the most important component in a computer?
2. What number does a bit that is turned on represent? What number does a bit that is turned off represent?
3. What would you call a device that works with binary data?
4. What are the words that make up a high-level programming language called?
5. What are the short words that are used in assembly language called?
6. What is the difference between a compiler and an interpreter?
7. What type of software controls the internal operations of the computer's hardware?

Exercises

1. To make sure that you can interact with the Python interpreter, try the following steps on your computer:

   - Start the Python interpreter in interactive mode.
   - At the >>> prompt type the following statement and then press Enter:

     ```
     print 'This is a test of the Python interpreter.' [ENTER]
     ```

   - After pressing the Enter key the interpreter will execute the statement. If you typed everything correctly, your session should look like this:

     ```
     >>> print 'This is a test of the Python interpreter.' [ENTER]
     This is a test of the Python interpreter.
     >>>
     ```

   - If you see an error message, enter the statement again and make sure you type it exactly as shown.
   - Exit the Python interpreter. (In Windows, press Ctrl-Z followed by Enter. On other systems press Ctrl-D.)

2. To make sure that you can interact with IDLE, try the following steps on your computer:

   - Start IDLE. To do this in Windows, click the *Start* button, then *All Programs*. In the Python program group click *IDLE (Python GUI)*.
   - When IDLE starts, it should appear similar to the window previously shown in Figure 1-21. At the >>> prompt type the following statement and then press Enter:

     ```
     print 'This is a test of IDLE.' [ENTER]
     ```

   - After pressing the Enter key the Python interpreter will execute the statement. If you typed everything correctly, your session should look like this:

     ```
     >>> print 'This is a test of IDLE.' [ENTER]
     This is a test of IDLE.
     >>>
     ```

   - If you see an error message, enter the statement again and make sure you type it exactly as shown.
   - Exit IDLE by clicking File, then Exit (or pressing Ctrl-Q on the keyboard).

3. Use what you've learned about the binary numbering system in this chapter to convert the following decimal numbers to binary:

   11
   65
   100
   255

4. Use what you've learned about the binary numbering system in this chapter to convert the following binary numbers to decimal:

   ```
   1101
   1000
   101011
   ```

5. Look at the ASCII chart in Appendix C and determine the codes for each letter of your first name.
6. Use the Internet to research the history of the Python programming language, and answer the following questions:
    - Who was the creator of Python?
    - When was Python created?
    - In the Python programming community, the person who created Python is commonly referred to as the "BDFL." What does this mean?

CHAPTER 2

Input, Processing, and Output

TOPICS

2.1 Designing a Program
2.2 Input, Processing, and Output
2.3 Displaying Output with the **`print`** Statement
2.4 Comments
2.5 Variables
2.6 Reading Input *from* the Keyboard
2.7 Performing Calculations
2.8 More About Data Output

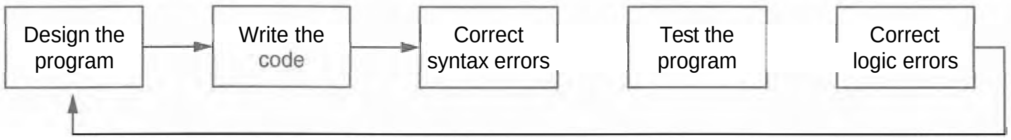
2.1 Designing a Program

— CONCEPT: Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

The Program Development Cycle

In Chapter 3 you learned that programmers typically use high-level languages such as Python to create programs. There is much more to creating a program than writing code, however. The process of creating a program that works correctly typically requires the five phases shown in Figure 2-1. The entire process is known as the *program development cycle.*

Figure 2-1 The program development cycle



Let's take a closer look at each stage in the cycle.

1. Design the Program All professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin a new project, they never jump right in and start writing code as the first step. They start by creating a design of the program. There are several ways to design a program, and later in this section we will discuss some techniques that you can use to design your Python programs.
2. Write the Code After designing the program, the programmer begins writing code in a high-level language such as Python. Recall from Chapter 1 that each language has its own rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.
3. Correct Syntax Errors If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).
4. Test the Program Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A logic *error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)
5. Correct Logic Errors If the program produces incorrect results, the programmer *debugs* the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over, and continues until no errors can be found.

More About the Design Process

The process of designing a program is arguably the most important part of the cycle. You can think of a program's design as its foundation. If you build a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.

The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform.
2. Determine the steps that must be taken to perform the task.

Let's take a closer look at each of these steps.

Understand the Task That the Program Is to Perform

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term *customer* to describe the person, group, or organization that is asking you to write a program. This could be a customer in the traditional sense of the word, meaning someone who is paying you to write a program. It could also be your boss, or the manager of a department within your company. Regardless of whom it is, the customer will be relying on your program to perform an important task.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A software requirement is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

TIP: If you choose to become a professional software developer, your customer will be anyone who asks you to write programs as part of your job. As long as you are a student, however, your customer is your instructor! In every programming class that you will take, it's practically guaranteed that your instructor will assign programming problems for you to complete. For your academic success, make sure that you understand your instructor's requirements for those assignments and write your programs accordingly.

Determine the Steps That Must Be Taken to Perform the Task

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. This is similar to the way you would break down a task into a series of steps that another person can follow. For example, suppose someone asks you how to boil water. You might break down that task into a series of steps as follows:

1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

This is an example of an algorithm, which is a set of well-defined logical steps that must be taken to perform a task. Notice that the steps in this algorithm are sequentially ordered. Step 1 should be performed before step 2, and so on. If a person follows these steps exactly as they appear, and in the correct order, he or she should be able to boil water successfully.

A programmer breaks down the task that a program must perform in a similar way. An algorithm is created, which lists all of the logical steps that must be taken. For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here are the steps that you would take:

1. Get the number of hours worked.
2. Get the hourly pay rate.

3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in steps 3.

Of course, this algorithm isn't ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts. Let's look at each of these in more detail.

Pseudocode

Because small mistakes like misspelled words and forgotten punctuation characters can cause syntax errors, programmers have to be mindful of such small details when writing code. For this reason, programmers find it helpful to write a program in pseudocode (pronounced "sue doe code") before they write it in the actual code of a programming language such as Python.

The word "pseudo" means fake, so *pseudocode* is fake code. It is an informal language that has no syntax rules, and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models, or "mock-ups" of programs. Because programmers don't have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code. Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:

```
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
```

Each statement in the pseudocode represents an operation that can be performed in Python. For example, Python can read input that is typed on the keyboard, perform mathematical calculations, and display messages on the screen.

Flowcharts

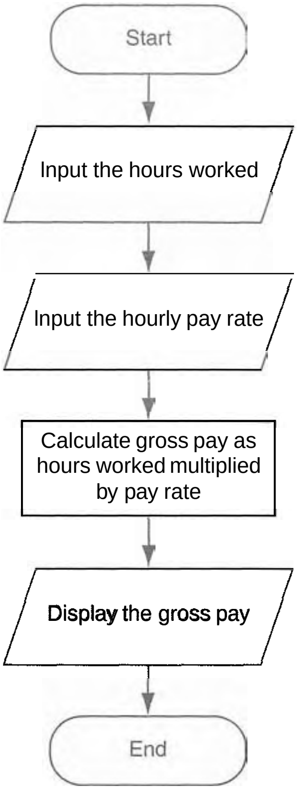
Flowcharting is another tool that programmers use to design programs. A *flowchart* is a diagram that graphically depicts the steps that take place in a program. Figure 2-2 shows how you might create a flowchart for the pay calculating program.

Notice that there are three types of symbols in the flowchart: ovals, parallelograms, and a rectangle. Each of these symbols represents a step in the program, as described here:

- The ovals, which appear at the top and bottom of the flowchart, are called *terminal symbols*. The *Start* terminal symbol marks the program's starting point and the *End* terminal symbol marks the program's ending point.
- Parallelograms are used as *input symbols* and *output symbols*. They represent steps in which the program reads input or displays output.
- Rectangles are used as *processing symbols*. They represent steps in which the program performs some process on data, such as a mathematical calculation.

The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal.

Figure 2-2 Flowchart for the pay calculating program


Start
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
End

Checkpoint

2.1 Who is a programmer's customer?

2.2 What is a software requirement?

2.3 What is an algorithm?

2.4 What is pseudocode?

2.5 What is a flowchart?

2.6 What do each of the following symbols mean in a flowchart?
- Oval
- Parallelogram
- Rectangle

2.2 Input, Processing, and Output

CONCEPT: Input is data that the program receives. When a program receives data, it usually processes it by performing some operation with it. The result of the operation is sent out of the program as output.

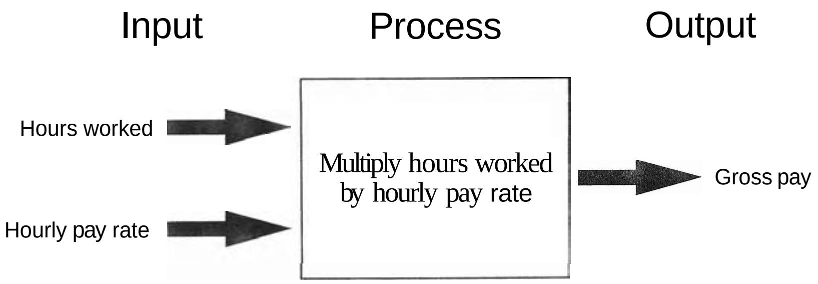
Computer programs typically perform the following three-step process:

1. Input is received.
2. Some process is performed on the input.
3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

Figure 2-3 illustrates these three steps in the pay calculating program that we discussed earlier. The number of hours worked and the hourly pay rate are provided as input. The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

Figure 2-3 The input, processing, and output of the pay calculating program



In this chapter we will discuss basic ways that you can perform input, processing, and output using Python.

2.3 Displaying Output with the `print` Statement

CONCEPT: You use the `print` statement to display output in a Python program.

Perhaps the most fundamental thing that a program can do is display a message on the computer screen. As you saw in Chapter 1, the `print` statement in Python displays output on the screen. Here is an example:

```
print 'Hello world'
```

The purpose of this statement is to display the message *Hello world* on the screen. Notice that after the word `print`, we have written **`Hello world`** inside single-quote marks. The quote marks will not be displayed when the statement executes. They simply mark the beginning and the end of the text that we wish to display.

Suppose your instructor tells you to write a program that displays your name and address on the computer screen. Program 2-1 shows an example of such a program, with the output that it will produce when it runs. (The line numbers that appear in a program listing in this book are *not* part of the program. We use the line numbers in our discussion to refer to parts of the program.)

Program 2-1 (output.py)

```
print 'Kate Austen'
print '123 Dharma Lane'
print 'Asheville, NC 28899'
```

Program Output

```
Kate Austen
123 Dharma Lane
Asheville, NC 28899
```

It is important to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. When you run this program, the first statement will execute, followed by the second statement, and followed by the third statement.

Strings and String Literals

Programs almost always work with data of some type. For example, Program 2-1 uses the following three pieces of data:

```
'Kate Austen'
'123 Dharma Lane'
'Asheville, NC 28899'
```

These pieces of data are sequences of characters. In programming terms, a sequence of characters that is used as data is called a *string*. When a string appears in the actual code of a program it is called a *string literal*. In Python code, string literals must be enclosed in quote marks. As mentioned earlier, the quote marks simply mark where the string data begins and ends.

In Python you can enclose string literals in a set of single-quote marks(') or a set of double-quote marks ("). The string literals in Program 2-1 are enclosed in single-quote marks, but the program could also be written as shown in Program 2-2.

Program 2-2 (double_quotes.py)

```
print "Kate Austen"
print "123 Dharma Lane"
print "Asheville, NC 28899"
```

Program Output

```
Kate Austen
123 Dharma Lane
Asheville, NC 18899
```

If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks. For example, Program 2-3 prints two strings that contain apostrophes.

Program 2-3 (apostrophe.py)

```
print "Don't fear!"
print "I'm here!"
```

Program Output

```
Don't fear!
I'm here!
```

Likewise, you can use single-quote marks to enclose a string literal that contains double-quotes as part of the string. Program 2-4 shows an example.

Program 2-4 (display_quote.py)

```
print 'Your assignment is to read "Hamlet" by tomorrow.'
```

Program Output

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python also allows you to enclose string literals in triple quotes (either `"""` or `'''`). Triple-quoted strings can contain both single quotes and double quotes as part of the string. The following statement shows an example:

```
print """I'm reading "Hamlet" tonight."""
```

This statement will print

```
I'm reading "Hamlet" tonight.
```

Triple quotes can also be used to surround multiline strings, something for which single and double quotes cannot be used. Here is an example:

```
print """"One
Two
Three """
```

This statement will print

```
One
Two
Three
```

Checkpoint

2.7 Write a `print` statement that displays your name.

2.8 Write a `print` statement that displays the following text:

```
Python's the best!
```

2.9 Write a `print` statement that displays the following text:

```
The cat said "meow."
```

2.4 Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the Python interpreter ignores them. They are intended for people who may be reading the source code.

Comments are short notes placed in different parts of a program, explaining how those parts of the program work. Although comments are a critical part of a program, they are ignored by the Python interpreter. Comments are intended for any person reading a program's code, not the computer.

In Python you begin a comment with the # character. When the Python interpreter sees a # character, it ignores everything from that character to the end of the line. For example, look at Program 2-5. Lines 1 and 2 are comments that briefly explain the program's purpose.

Program 2-5 (comment1.py)

```
1  # This program displays a person's
2  # name and address.
3  print 'Kate Austen'
4  print '123 Dharma Lane'
5  print 'Asheville, NC 28899'
```

Program Output

```
Kate Austen
123 Dharma Lane
Asheville, NC 28899
```

Programmers commonly write end-line comments in their code. An *end-line comment* is a comment that appears at the end of a line of code. It usually explains the statement that appears in that line. Program 2-6 shows an example. Each line ends with a comment that briefly explains what the line does.

Program 2-6 (comment2.py)

```
print "Kate Austen"         # Display the name
print "123 Dharma Lane"     # Display the street address
print "Asheville, NC 28899" # Display the city, state, and ZIP
```

Program Output

```
Kate Austen
123 Dharma Lane
Asheville, NC 28899
```

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it can seem more productive to write code that actually does something! It is crucial that you take the extra time to write comments, however. They will almost certainly save you and others time in the future when you have to modify or debug the program. Large and complex programs can be almost impossible easy to read and understand if they are not properly commented.

2.5 Variables

CONCEPT: **A** variable is a name that represents a value stored in the computer's memory.

Programs usually store data in the computer's memory and perform operations on that data. For example, consider the typical online shopping experience: you browse a website and add the items that you want to purchase to the shopping cart. As you add items to the shopping cart, data about those items is stored in memory. Then, when you click the checkout button, a program running on the website's computer calculates the cost of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and the total of all these charges. When the program performs these calculations, it stores the results in the computer's memory.

Programs use variables to access and manipulate data that is stored in memory. A *variable* is a name that represents a value in the computer's memory. For example, a program that calculates the sales tax on a purchase might use the variable name `tax` to represent that value in memory. And a program that calculates the distance between two cities might use the variable name `distance` to represent that value in memory. When a variable represents a value in the computer's memory, we say that the variable *references* the value.

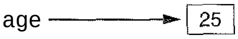
Creating Variables with Assignment Statements

You use an *assignment statement* to create a variable in Python. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named `age` will be created and it will reference the value 25. This concept is shown in Figure 2-4. In the figure, think of the value 25 as being stored somewhere in the computer's memory. The arrow that points from `age` to the value 25 indicates that the name `age` references the value.

Figure 2-4 The age variable references the value 25



An assignment statement is written in the following general format:

```
variable = expression
```

The equal sign (=) is known as the *assignment operator*. In the general format, *`variable`* is the name of a variable and `expression` is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

In an assignment statement, the variable that is receiving the assignment must appear on the left side of the = operator. For example, the following statement will cause an error:

```
25 = age    # This is an error!
```

The code in Program 2-7 demonstrates a variable. Line 2 creates a variable named room and assigns it the value 503. The `print` statements in lines 3 and 4 display a message. Notice that line 4 displays the value that is referenced by the room variable.

Program 2-7 (variable_demo.py)

```
1  # This program demonstrates a variable.
2  room = 503
3  print 'I am staying in room number'
4  print room
```

Program Output

```
I [illegible] room number
503
```

Notice that in line 4 there are no quotation marks around room. If quotation marks were placed around room, it would have indicated that we want to display the word "room" instead of the contents of the room variable. In other words, the following statement will display the contents of the room variable:

```
print room
```

This statement, however, will display the word "age":

```
print 'age'
```

Program 2-8 shows a sample program that uses two variables. Line 2 creates a variable named `top_speed`, assigning it the value 160. Line **3** creates a variable named `distance`, assigning it the value 300. This is illustrated in Figure *2-5*.

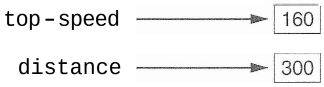
Program 2-8 (variable_demo2.py)

```
1  # Create two variables: top-speed and distance.
2  top-speed = 160
3  distance = 300
4
5  # Display the values referenced by the variables.
6  print 'The top speed is'
7  print top-speed
8  print 'The distance traveled is'
9  print distance
```

Program Output

```
The top speed is
160
The distance traveled is
300
```

Figure *2-5* Two variables



WARNING! You cannot use a variable until you have assigned a value to it. An error will occur if you try to perform an operation on a variable, such as printing it, before it has been assigned a value.

Sometimes a simple typing mistake will cause this error. One example is a misspelled variable name, as shown here:

```
temperature = 74.5  # Create a variable
print tempereture   # Error! Misspelled variable name
```

In this code, the variable `temperature` is created by the assignment statement. The variable name is spelled differently in the `print` statement, however, which will cause an error. Another example is the inconsistent use of uppercase and lowercase letters in a variable name. Here is an example:

```
temperature = 74.5  # Create a variable
print Temperature   # Error! Inconsistent use of case
```

In this code the variable `temperature` (in all lowercase letters) is created by the assignment statement. In the `print` statement, the name `Temperature` is spelled with an uppercase T. This will cause an error because variable names are case sensitive in Python.

Variable Naming Rules

Although you are allowed to make up your own names for variables, you must follow these rules:

- You cannot use one of Python's key words as a variable name. (SeeTable 1-2 for a list of the key words.)
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as **`itemsordered.`**

In addition to following these rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named **`temperature,`** and a variable that holds a car's speed might be named **`speed.`** You may be tempted to give variables names like `x` and `b2`, but names like these give no clue as to what the variable's purpose is.

Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following variable names:

```
grosspay
payrate
hotdogssoldtoday
```

Unfortunately, these names are not easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name, and make it more readable to the human eye.

One way to do this is to use the underscore character to represent a space. For example, the following variable names are easier to read than those previously shown:

```
gross_pay
pay-rate
hot-dogs-sold-today
```

This style of naming variables is popular among Python programmers and is the style we will use in this book. There are other popular styles, however, such as the *camelCase* naming convention. camelCase names are written in the following manner:

- The variable name begins with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

NOTE: This style of naming is called camelCase because the uppercase characters that appear in a name may suggest a camel's humps.

Table 2-1 lists several sample variable names and indicates whether each is legal or illegal in Python.

Table 2-1 Sample variable names

Variable Name	Legal or Illegal?
`units—per—day`	Legal
`dayOfWeek`	Legal
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`June1997`	Legal
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

Displaying Multiple Items with the `print` Statement

If you refer to Program 2-7 you will see that we used the following two `print` statements in lines **3** and 4:

```
print 'I am staying in room number'
print room
```

We used two `print` statements because we needed to display two pieces of data. Line **3** displays the string literal `'I am staying in room number'`, and line 4 displays the value referenced by the `room` variable.

This program can be simplified, however, because Python allows us to display multiple items with one `print` statement. We simply have to separate the items with commas as shown in Program 2-9.

Program 2-9 (variable_demo3.py)

```
1  # This program demonstrates a variable.
2  room = 503
3  print 'I am staying in room number', room
```

Program Output

```
I am staying in room number 503
```

The `print` statement in line 4 displays two items: a string literal followed by the value referenced by the `room` variable. Notice that Python automatically printed a space between these two items. When multiple items are printed in one line of output, they will automatically be separated by a space.

Variable Reassignment

Variables are called "variable" because they can reference different values while a program is running. When you assign a value to a variable, the variable will reference that value until you assign it a different value. For example, look at Program 2-10. The statement in line **3** creates a variable named **`dollars`** and assigns it the value 2.75. This is shown in the top part of Figure 2-6. Then, the statement in line 8 assigns a different value, 99.95, to the **`dollars`** variable. The bottom part of Figure 2-6 shows how this changes the **`dollars`** variable. The old value, 2.75, is still in the computer's memory, but it can no longer be used because it isn't referenced by a variable. (The Python interpreter will eventually remove the unusable value from memory.)

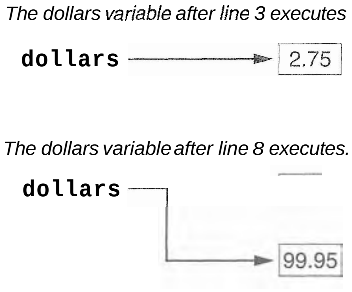
Program 2-10 (variable_demo4.py)

```
1  # This program demonstrates variable reassignment.
2  # Assign a value to the dollars variable.
3  dollars = 2.75
4  print 'I have', dollars, 'in my account.'
5
6  # Reassign dollars so it references
7  # a different value.
8  dollars = 99.95
9  print 'But now I have', dollars, 'in my account!'
```

Program Output

```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

Figure *2-6* Variable reassignment in Program 2-10


The dollars variable after line 3 executes
dollars
2.75
The dollars variable after line 8 executes.
dollars
99.95

Numeric Data Types and Literals

In Chapter 1 we discussed the way that computers store data in memory. (See section 1.3) You might recall from that discussion that computers use a different technique for storing real numbers (numbers with a fractional part) than for storing integers. Not only are these types of numbers stored differently in memory, but similar operations on them are carried out in different ways.

Because different types of numbers are stored and manipulated in different ways, Python uses *data types* to categorize values in memory. When an integer is stored in memory, it is classified as an `int`, and when a real number is stored in memory, it is classified as a `float`.

Let's look at how Python determines the data type of a number. Several of the programs that you have seen so far have numeric data written into their code. For example, the following statement, which appears in Program 2-9, has the number 503 written into it.

```
room = 503
```

This statement causes the value 503 to be stored in memory, and it makes the room variable reference it. The following statement, which appears in Program 2-10, has the number 2.75 written into it.

```
dollars = 2.75
```

This statement causes the value 2.75 to be stored in memory, and it makes the `dollars` variable reference it. A number that is written into a program's code is called a *numeric literal*. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an `int`. Examples are `7`, `124`, and `-9`.
- A numeric literal that is written with a decimal point is considered a `float`. Examples are `1.5`, `3.14159`, and `5.0`.

So, the following statement causes the number 503 to be stored in memory as an `int`:

```
room = 503
```

And the following statement causes the number 2.75 to be stored in memory as a `float`:

```
dollars = 2.75
```

When you store an item in memory, it is important for you to be aware of the item's data type. As you will see, some operations behave differently depending on the type of data involved, and some operations can only be performed on values of a specific data type.

WARNING! You cannot write currency symbols, spaces, or commas in numeric literals. For example, the following statement will cause an error:

```
value = $4,567.99 # Error!
```

This statement must be written as:

```
value = 4567.99 # Correct
```

Storing Strings with the `str` Data Type

In addition to the int and `float` data types, Python also has a data type named `str`, which is used for storing strings in memory. The code in Program 2-11 shows how strings can be assigned to variables.

Program 2-11 (string_variable.py)

```
# Create variables to reference two strings.
first-name = 'Kathryn'
last_name = 'Marino'

# Display the values referenced by the variables.
print first-name, last-name
```

Program Output

```
Kathryn Marino
```

Checkpoint

2.10 What is a variable?

2.11 Which of the following are illegal variable names in Python, and why?

```
x
99bottles
july2009
theSalesFigureForFiscalYear
r&d
grade-report
```

2.12 Is the variable name `Sales` the same as `sales`? Why or why not?

2.13 Is the following assignment statement valid or invalid? If it is invalid, why?

```
72 = amount
```

2.14 What will the following code display?

```
val = 99
print 'The value is', 'val'
```

2.15 Look at the following assignment statements:

```
value1 = 99
value2 = 45.9
value3 = 7.0
value4 = 7
value5 = 'abc'
```

After these statements execute, what is the Python data type of the values referenced by each variable?

2.16 What will be displayed by the following program?

```
my-value = 99
my-value = 0
print my-value
```

2.6 Reading Input from the Keyboard

CONCEPT: Programs commonly need to read input typed by the user on the keyboard. We will use the Python functions to do this.

Most of the programs that you will write will need to read input, and then perform an operation on that input. In this section, we will discuss a basic input operation: reading data that has been typed on the keyboard. When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program.

In this book we will use two of Python's built-in functions to read input from the keyboard. A *function* is a piece of prewritten code that performs an operation and then returns a value back to the program. We will use the `input` function to read numeric data from the keyboard, and the `raw_input` function to read strings as input.

Reading Numbers with the `input` Function

Python's `input` function is useful for reading numeric input from the keyboard. You normally use the `input` function in an assignment statement that follows this general format:

```
variable = input(prompt)
```

In the general format, `prompt` is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value. `variable` is the name of a variable that will reference the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

```
hours = input('How many hours did you work? ')
```

When this statement executes, the following things happen:

- The string `'How many hours did you work? '` is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard, and then press the Enter key.
- When the Enter key is pressed, the data that was typed is assigned to the `hours` variable.

Program 2-12 shows a sample program that uses the `input` function.

Program 2-12 (input.py)

```
1  # This program gets input from the user.
2  age = input('How old are you? ')
3  print 'You said that you are', age, 'years old.'
```

Program Output (with input shown in bold)

```
How old are you? 28 [Enter]
You said that you are 28 years old.
```

The statement in line 2 uses the `input` function to read data that is typed on the keyboard. In the sample run, the user typed 28 and then pressed Enter. As a result, the integer value 28 was assigned to the `age` variable.

Take a closer look at the string we used as a prompt, in line 2:

```
'How old are you? '
```

Notice that the last character in the string, inside the quote marks, is a `space`. We put a space there because the `input` function does not automatically display a space after the prompt. When the user begins typing characters, they will appear on the screen immediately after the prompt. Making the last character in the prompt a space visually separates the prompt from the user's input on the screen.

When the user enters a number in response to the `input` function, Python determines the number's data type in the same way that it determines a numeric literal's data type: If the number contains no decimal point it is stored in memory as an `int`. If it contains a decimal point it is stored in memory as a `float`.

NOTE: In this section, we mentioned the user. The *user* is simply any hypothetical person that is using a program and providing input for it. The user is sometimes called the end *user*.

Reading Strings with the `raw_input` Function

Although the `input` function works well for reading numbers, it is not convenient for reading strings. In order for the `input` function to read data as a string, the user has to enclose the data in quote-marks when he or she types it on the keyboard. Most users are not accustomed to doing this, so it's best to use another function: `raw_input`.

The `raw_input` function works like the `input` function, with one exception: the `raw_input` function retrieves all keyboard input as a string. There is no need for the user to type quote marks around the data that is entered. Program 2-13 shows a sample program that uses the `raw_input` function to read strings.

Program 2-13 (string_input.py)

```
# Get the user's first name.
first_name = raw_input('Enter your first name: ')

# Get the user's last name.
last_name = raw_input('Enter your last name: ')

# Print a greeting to the user.
print 'Hello', first_name, last_name
```

Program Output (with input shown in bold)

```
Enter your first name: Vinny [Enter]
Enter your last name: Brown [Enter]
Hello Vinny Brown
```

Checkpoint

2.17 You need the user of a program to enter the amount of sales for the week. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.18 You need the user of a program to enter a customer's last name. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.7 Performing Calculations

CONCEPT: Python has numerous operators that can be used to perform mathematical calculations.

Most real-world algorithms require calculations to be performed. A programmer's tools for performing calculations are math operators. Table 2-2 lists the math operators that are provided by the Python language.

Table 2-2 Python math operators

Symbol	Operation	Description
`+`	Addition	Adds two numbers
`-`	Subtraction	Subtracts one number from another
`*`	Multiplication	Multiplies one number by another
`/`	Division	Divides one number by another and gives the quotient
`%`	Remainder	Divides one number by another and gives the remainder
`**`	Exponent	Raises a number to a power

Programmers use the operators shown in Table 2-2 to create math expressions. A *math expression* performs a calculation and gives a value. The following is an example of a simple math expression:

```
12 + 2
```

The values on the right and left of the + operator are called operands. These are values that the + operator adds together. The value that is given by this expression is 14.

Variables may also be used in a math expression. For example, suppose we have two variables named `hours` and `pay_rate`. The following math expression uses the `*` operator to multiply the value referenced by the `hours` variable by the value referenced by the `pay_rate` variable:

```
hours * pay-rate
```

When we use a math expression to calculate a value, normally we want to save that value in memory so we can use it again in the program. We do this with an assignment statement. Program 2-14 shows an example.

Program 2-14 (simple_math.py)

```
 1  # Assign a value to the salary variable.
 2  salary = 2500.0
 3
 4  # Assign a value to the bonus variable.
 5  bonus = 1200.0
 6
 7  # Calculate the total pay by adding salary
 8  # and bonus. Assign the result to pay.
 9  pay = salary + bonus
10
11  # Display the pay.
12  print 'Your pay is', pay
```

Program Output

```
Your pay is 3700.0
```

Line 2 assigns 2500.0 to the `salary` variable, and line 5 assigns 1200.0 to the `bonus` variable. Line 9 assigns the result of the expression `salary + bonus` to the `pay` variable. As you can see from the program output, the `pay` variable holds the value 3700.0.

In the Spotlight:

Calculating a Percentage

Determining percentages is a common calculation in computer programming. In mathematics, the % symbol is used to indicate a percentage, but most programming languages, Python included, do not use the % symbol for this purpose. In a program, you usually have to convert a percentage to a decimal number. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's step through the process of writing a program that calculates a percentage. Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

1. *Get the original price of the item.*
2. *Calculate 20 percent of the original price. This is the amount of the discount.*
3. *Subtract the discount from the original price. This is the sale price.*
4. *Display the sale price.*

In step 1 we get the original price of the item. We will prompt the user to enter this data on the keyboard. In our program we will use the following statement to do this. Notice that the value entered by the user will be stored in a variable named `original_price`.

```
original_price = input("Enter the item's original price: ")
```

In step 2, we calculate the amount of the discount. To do this we multiply the original price by 20 percent. The following statement performs this calculation and assigns the result to the `discount` variable.

```
discount = original_price * 0.2
```

In step 3, we subtract the discount from the original price. The following statement does this calculation and stores the result in the `sale_price` variable.

```
sale_price = original_price – discount
```

Last, in step 4, we will use the following statement to display the sale price:

```
print 'The sale price is', sale_price
```

Program 2-15 shows the entire program, with example output.

Program 2-15 (sale_price.py)

```
# This program gets an item's original price and
# calculates its sale price, with a 20% discount.

# Get the item's original price.
original_price = input("Enter the item's original price: ")

# Calculate the amount of the discount.
discount = original_price * 0.2

# Calculate the sale price.
sale_price = original_price - discount

# Display the sale price.
print 'The sale price is', sale_price
```

Program Output (with input shown in bold)

```
Enter the item's original price: 100.00 [Enter]
The sale price is 80.0
```

Integer Division

Be careful when dividing an integer by another integer. In Python, as well as many other languages, when an integer is divided by an integer the result will also be an integer. This behavior is known as integer ***division.*** For example, look at the following statement:

```
number = 3 / 2
```

What value will the `number` reference after this statement executes? You would probably assume that `number` would reference the value 1.5 because that's the result your calculator shows when you divide 3 by 2. However, that's not what will happen. Because the numbers **3** and 2 are both treated as integers, Python will throw away the fractional part of the result. (Throwing away the fractional part of a number is called *truncation.)* As a result, the statement will assign the value 1 to the `number` variable, not 1.5.

If you want to make sure that a division operation yields a real number, at least one of the operands must be a number with a decimal point or a `float` variable. For example, we could rewrite the statement as follows:

```
number = 3.0 / 2.0
```

Operator Precedence

You can write statements that use complex mathematical expressions involving several operators. The following statement assigns the sum of 17, the variable x, 21, and the variable y to the variable `answer`.

```
answer = 17 + x + 21 + y
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12.0 + 6.0 / 3.0
```

What value will be assigned to `outcome`? The number 6.0 might be used as an operand for either the addition or division operator. The `outcome` variable could be assigned either 6.0 or 14.0, depending on when the division takes place. Fortunately, the answer can be predicted because Python follows the same order of operations that you learned in math class.

First, operations that are enclosed in parentheses are performed first. Then, when two operators share an operand, the operator with the higher *precedence* is applied first. The precedence of the math operators, from highest to lowest, are:

1. Exponentiation: `**`
2. Multiplication, division, and remainder: `* / %`
3. Addition and subtraction: `+ -`

Notice that the multiplication (`*`), division (`/`), and remainder (`%`) operators have the same precedence. The addition (`+`) and subtraction (`-`) operators also have the same precedence. When two operands with the same precedence share an operand, the operators execute from left to right.

Now, let's go back to the previous math expression:

```
outcome = 12.0 + 6.0 / 3.0
```

The value that will be assigned to `outcome` is 14.0 because the division operator has a higher *precedence* than the addition operator. As a result, the division takes place before the addition. The expression can be diagrammed as shown in Figure 2-7.

Figure 2-7 Operator precedence

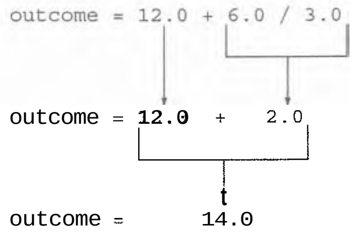


Table 2-3 shows some other sample expressions with their values.

Table 2-3 Some expressions

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`6 - 3 * 2 + 7 - 1`	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables a and b are added together, and their sum is divided by 4:

```
result = (a + b) / 4
```

Without the parentheses, however, b would be divided by 4 and the result added to a. Table 2-4 shows more expressions and their values.

Table 2-4 More expressions and their values

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(6 - 3) * (2 + 7) / 3`	9

In the Spotlight:

Calculating an Average

Determining the average of a group of values is a simple calculation: add all of the values and then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that the variables `a`, `b`, and `c` each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```
average = a + b + c / 3.0
```

Can you see the error in this statement? When it executes, the division will take place first. The value in `c` will be divided by 3, and then the result will be added to `a + b`. That is not the correct way to calculate an average. To correct this error we need to put parentheses around `a + b + c`, as shown here:

```
average = (a + b + c) / 3.0
```

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. *Get the first test score.*
2. *Get the second test score.*
3. *Get the third test score.*
4. *Calculate the average by adding the three test scores and dividing the sum by 3.*
5. *Display the average.*

In steps 1, 2, and 3 we will prompt the user to enter the three test scores. We will store those test scores in the variables `test1`, `test2`, and `test3`. In step 4 we will calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable:

```
average = (test1 + test2 + test3) / 3.0
```

Last, in step 5, we display the average. Program 2-16 shows the program.

Program 2-16 (test_score_average.py)

```
# Get three test scores and assign them to the
# test1, test2, and test3 variables.
test1 = input('Enter the first test score: ')
test2 = input('Enter the second test score: ')
test3 = input('Enter the third test score: ')

# Calculate the average of the three scores
# and assign the result to the average variable.
average = (test1 + test2 + test3) / 3.0

# Display the average.
print 'The average score is', average
```

Program Output (with input shown in bold)

```
Enter the first test score: 90 [Enter]
Enter the second test score: 80 [Enter]
Enter the third test score: 100 [Enter]
The average score is 90.0
```

The Exponent and Remainder Operators

In addition to the basic math operators for addition, subtraction, multiplication, and division, Python also provides an exponent operator and a remainder operator. Two asterisks written together (`**`) is the exponent operator, and its purpose it to raise a number to a power. For example, the following statement raises the `length` variable to the power of 2 and assigns the result to the `area` variable:

```
area = length**2
```

In Python, the `%` symbol is the remainder operator. (This is also known as the modulus operator.) The remainder operator performs division, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to `leftover`:

```
leftover = 17 % 3
```

This statement assigns 2 to `leftover` because 17 divided by 3 is 5 with a remainder of 2. You will not use the remainder operator frequently, but it is useful in some situations. It is commonly used in calculations that detect odd or even numbers, determine the day of the week, measure the passage of time, and other specialized operations.

Converting Math Formulas to Programming Statements

You probably remember from algebra class that the expression $2xy$ is understood to mean 2 times x times y. In math, you do not always use an operator for multiplication. Python, as well as other programming languages, requires an operator for any mathematical operation. Table 2-5 shows some algebraic expressions that perform multiplication and the equivalent programming expressions.

Table 2-5 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
$(3)(12)$	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
x = (a + b) / c
```

Table 2-6 shows additional algebraic expressions and their Python equivalents.

Table 2-6 Algebraic and programming expressions

Algebraic Expression	Python Statement
$y = 3\frac{x}{2}$	`y = 3 * x / 2`
$z = 3bc + 4$	`z = 3 * b * c + 4`
$a = \frac{x + 2}{b - 1}$	`a = (x + 2) / (b - 1)`

In the Spotlight:

Converting a Math Formula to a Programming Statement

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is $10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be convenient to write a computer program to perform the calculation, because then we can experiment with different values for the variables. Here is an algorithm that we can use:

1. *Get the desired future value.*
2. *Get the annual interest rate.*
3. *Get the number of years that the money will sit in the account.*
4. *Calculate the amount that will have to be deposited.*
5. *Display the result of the calculation in step 4.*

In steps 1 through 3, we will prompt the user to enter the specified values. We will assign the desired future value to a variable named `future_value`, the annual interest rate to a variable named `rate`, and the number of years to a variable named `years`.

In step 4, we calculate the present value, which is the amount of money that we will have to deposit. We will convert the formula previously shown to the following statement. The statement stores the result of the calculation in the `present_value` variable.

```
present_value = future_value / (1.0 + rate)**years
```

In step 5, we display the value in the `present_value` variable. Program 2-17 shows the program.

Program 2-17 (future_value.py)

```
# Get the desired future value.
future_value = input('Enter the desired future value: ')

# Get the annual interest rate.
rate = input('Enter the annual interest rate: ')

# Get the number of years that the money will appreciate.
years = input('Enter the number of years the money will grow: ')

# Calculate the amount needed to deposit.
present_value = future_value / (1.0 + rate)**years

# Display the amount needed to deposit.
print 'You will need to deposit this amount:', present_value
```

Program Output (with input shown in bold)

```
Enter the desired future value: 10000.0 [Enter]
Enter the annual interest rate: 0.05 [Enter]
Enter the number of years the money will grow: 10 [Enter]
You will need to deposit this amount: 6139.13253541
```

NOTE: Unlike the output shown for this program, dollar amounts are usually rounded to two decimal places. Later in this chapter you will learn how to format numbers so they are rounded to a specified number of decimal places.

Data Type Conversion

When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands. Python follows these rules when evaluating mathematical expressions:

- When an operation is performed on two **`int`** values, the result will be an **`int`**.
- When an operation is performed on two **`float`** values, the result will be a **`float`**.

When an operation is performed on an `int` and a `float`, the `int` value will be temporarily converted to a `float` and the result of the operation will be a `float`. (An expression that uses an `int` and a `float` is called a *mixed-type expression.)*

The first two situations are straightforward: operations on `ints` produce `ints`, and operations on `floats` produce `floats`. Let's look at an example of the third situation, which involves mixed-type expressions:

```
my-number = 5 * 2.0
```

When this statement executes, the value 5 will be converted to a `float` (5.0)and then multiplied by 2.0. The result, 10.0, will be assigned to my—number.

The `int` to `float` conversion that takes place in the previous statement happens implicitly. In some situations, you want to explicitly make sure that a value is converted to a specific type. For example, look at Program 2-18.

Program 2-18 (books-per-month1.py)

```
 1  # Get the number of books the user plans to read.
 2  books = input('How many books do you want to read? ')
 3
 4  # Get the number of months it will take to read them.
 5  months = input('How many months will it take? ')
 6
 7  # Calculate the number of books per month.
 8  per-month = books / months
 9
10  # Display the result.
11  print 'You will read', per-month, 'books per month.'
```

Program Output (with input shown in bold)

```
How [illegible] books do [illegible]ant to read? 20 [Enter]
How many months will it take? 8 [Enter]
You will read 2 books per month.
```

This program asks the user for the number of books he or she plans to read, and the number of months it will take to read them. Line 8 divides `books` by `months` to calculate the number of books that the user must read per month. However, if the user has entered integer values for both `books` and `months`, this statement will perform integer division. This was what happened in the sample output. If you want the result to be completely accurate, you need to make sure that at least one of the operands in the division operation is a `float`. We can do that with Python's built-in `float()` function, as shown in Program 2-19.

Program 2-19 (books_per_month2.py)

```
# Get the number of books the user plans to read.
books = input( 'How many books do you want to read? ')
```

(program continues)

Program 2-19 *(continued)*

```
 3
 4  # Get the number of months it will take to read them.
 5  months = input('How many months will it take? ')
 6
 7  # Calculate the number of books per month.
 8  per-month = float(books) / months
 9
10  # Display the result.
11  print 'You will read', per-month, 'books per month.'
```

Program Output (with input shown in bold)

```
How many books do you want to read? 20 [Enter]
How many months will it take? 8 [Enter]
You will read 2.5 books per month.
```

In line 8 the expression `float(books)` converts the value referenced by `books` to a `float`. This ensures that when the division takes place, one of the operands will be a `float`, thus preventing integer division.

> **WARNING!** Notice that in line 8 of Program 2-19, we did not put the entire expression `books / months` inside the parentheses of the `float` function, as shown here:
>
> ```
> per-month = float(books / months)
> ```
>
> This statement does not convert the value in `books` or `months` to a `float`, but converts the result of the expression `books / months`. If this statement were used in the program, an integer division operation would still have been performed. Here's why: The result of the expression `books / months` is 2 (because integer division takes place). The value 2 converted to a `float` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `float`.

Python also has a built-in `int()` function that converts a value to an `int`. When a `float` is converted to an `int`, any fractional part is thrown away, or truncated. Here is an example:

```
x = 27.9
y = int(x)
```

After this code executes, the variable `y` will be assigned 27. Here is an example showing the `int` function converting a negative `float` value:

```
x = -12.9
y = int(x)
```

After this code executes, `y` will be assigned –12.

Breaking Long Statements into Multiple Lines

Most programming statements are written on one line. If a programming statement is too long, however, you will not be able to view all of it in your editor window without scrolling horizontally. In addition, if you print your program code on paper and one of the statements is too long to fit on one line, it will wrap around to the next line and make the code difficult to read.

Python allows you to break a statement into multiple lines by using the line continuation character, which is a backslash (\). You simply type the backslash character at the point you want to break the statement, and then press the Enter key. Here is a `print` statement that is broken into two lines with the line continuation character:

```
print 'We sold', units-sold, \
      'for a total of', sales-amount
```

The line continuation character that appears at the end of the first line tells the interpreter that the statement is continued on the next line. Here is a statement that performs a mathematical calculation and has been broken up to fit on two lines:

```
result = varl * 2 + var2 * 3 + \
         var3 * 4 + var4 * 5
```

Here is one last example:

```
print "Monday's sales are", monday, \
      "and Tuesday's sales are", tuesday, \
      "and Wednesday's sales are", wednesday
```

This long `print` statement is broken into three lines. Notice that the first two lines end with a backslash.

Checkpoint

2.19 Complete the following table by writing the value of each expression in the Value column.

Expression	**Value**
`6 + 3 * 5`	_____
`12 / 2 - 4`	_____
`9 + 14 * 2 - 6`	_____
`(6 + 2) * 3`	_____
`14 / (11 - 4)`	_____
`9 + 12 * (8 - 3)`	_____
`float(9) / 2`	_____
`float(9 / 2)`	_____
`int(9.0 / 3.0)`	_____

2.20 What value will be assigned to `result` after the following statement executes?

```
result = 9 / 2
```

2.21 What value will be assigned to `result` after the following statement executes?

```
result = 9 % 2
```

2.8

More About Data Output

So far we have discussed only basic ways to display data. Eventually, you will want to exercise more control over the way data appear on the screen. In this section, you will learn more details about the Python `print` statement, and you'll see techniques for formatting output in specific ways.

Suppressing the `print` Statement" Newline

The `print` statement normally displays a line of output. For example, the following three print statements will produce three lines of output:

```
print 'One'
print 'Two'
print 'Three'
```

Each of the `print` statements shown here displays a string and then prints a *newline character*. You do not see the newline character, but when it is displayed, it causes the output to advance to the next line. (You can think of the newline character as a special command that causes the computer to start a new line of output.)

If you do not want the `print` statement to start a new line of output when it finishes displaying its output, you can write a trailing comma at the end of the statement, as shown here:

```
print 'One',
print 'Two',
print 'Three'
```

Notice that the first two `print` statements end with a comma. The trailing commas prevent these two `print` statements from displaying a newline character at the end of their output. Instead, they display a space at the end of their output. Here is the output of these statements:

```
One Two Three
```

Escape Characters

An *escape character* is a special character that is preceded with a backslash (`\`), appearing inside a string literal. When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

For example, `\n` is the newline escape character. When the `\n` escape character is printed, it isn't displayed on the screen. Instead, it causes output to advance to the next line. For example, look at the following statement:

```
print 'One\nTwo\nThree'
```

When this statement executes, it displays

```
One
Two
Three
```

Python recognizes several escape characters, some of which are listed in Table 2-7.

Table 2-7 *Some of Python's escape characters*

Escape Character	Effect
`\n`	Causes output to be advanced to the next line.
`\t`	Causes output to skip over to the next horizontal tab position.
`\'`	Causes a single quote mark to be printed.
`\"`	Causes a double quote mark to be printed.
`\\`	Causes a backslash character to be printed.

The `\t` escape character advances the output to the next horizontal tab position. (A tab position normally appears after every eighth character.) The following statements are illustrative:

```
print 'Mon\tTues\tWed'
print 'Thur\tFri\tSat'
```

This statement prints Monday, then advances the output to the next tab position, then prints Tuesday, then advances the output to the next tab position, then prints Wednesday. The output will look like this:

```
Mon    Tues   Wed
Thur   Fri    Sat
```

You can use the `\'` and `\"` escape characters to display quotation marks. The following statements are illustrative:

```
print "Your assignment is to read \"Hamlet\" by tomorrow."
print 'I\'m ready to begin.'
```

These statements display the following:

```
Your assignment is to read "Hamlet" by tomorrow.
I'm ready to begin.
```

You can use the \\ escape character to display a backslash, as shown in the following:

```
print 'The path is C:\\temp\\data.'
```

This statement will display

```
The path is C:\temp\data.
```

Displaying Multiple Items with the + Operator

Earlier in this chapter, you saw that the + operator is used to add two numbers. When the + operator is used with two strings, however, it performs string concatenation.

This means that it appends one string to another. For example, look at the following statement:

```
print 'This is ' + 'one string.'
```

This statement will print

```
This is one string.
```

String concatenation can be useful for breaking up a string literal so a long `print` statement can span multiple lines. Here is an example:

```
print 'Enter the amount of ' + \
      'sales for each day and ' + \
      'press Enter.'
```

This statement will display the following:

```
Enter the amount of sales for each day and press Enter.
```

Formatting Numbers

You might not always be happy with the way that numbers, especially floating-point numbers, are displayed on the screen. When a floating-point number is displayed by the `print` statement, it can appear with up to 12 significant digits. This is shown in the output of Program 2-20.

Program 2-20 (no_formatting.py)

```
# This program demonstrates how a floating-point
# number is displayed with no formatting.
amount-due = 5000.0
monthly-payment = amount-due / 12.0
print 'The monthly payment is', monthly-payment
```

Program Output

```
The monthly payment is 416.666666667
```

Because this program displays a dollar amount, it would be nice to see that amount rounded to two decimal places. Fortunately, Python gives us a way to do just that with the string format operator.

You previously learned that the `%` symbol is the remainder operator. That's true when both of its operands are numbers. When the operand on the left side of the `%` symbol is a string, however, it becomes the *string format operator*. Here is the general format of how we can use the string format operator with the `print` statement to format the way a number is displayed:

```
print string % number
```

In the general format, ***`string`*** is a string that contains text and/or a formatting specifier. A *formatting specifier* is a special set of characters that specify how a value should be formatted. In the general format, *number* is a variable or expression that gives a numeric value. The value of *number* will be formatted according to the formatting specifier in the *string*. Here is an example:

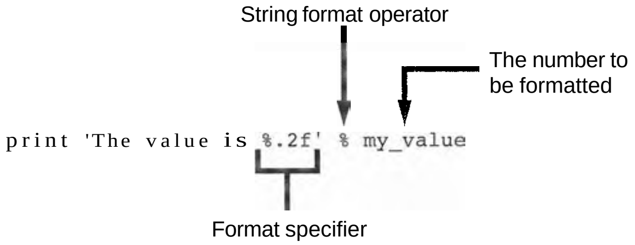
```
my - value = 7.23456
print 'The value is %.2f' % my - value
```

Figure 2-8 points out the important parts of the `print` statement. In the `print` statement, the formatting specifier is `%.2f`. When the statement executes, `%.2f` will not be displayed. Instead, the value referenced by `my_value` will be displayed in place of `%.2f`. Here is the way the output will appear:

```
The value is 7.23
```

The `f` in the formatting specifier indicates that we want to display a floating-point number. The `.2` that appears before the `f` indicates that the number should be rounded to two decimal places. Program 2-21 shows how we can modify Program 2-20 so that it formats its output using this technique.

Figure 2-8 Using the string format operator


String format operator
The number to be formatted
print 'The value is %.2f' % my_value
Format specifier

Program 2-21 (formatting.py)

```
# This program demonstrates how a floating-point
# number can be formatted.
amount - due = 5000.0
monthly - payment = amount - due / 12.0
print 'The monthly payment is %.2f' % monthlypayment
```

Program Output

```
The monthly payment is 416.67
```

You can round values to other numbers of decimal places. For example, the formatting specifier `%.3f` specifies three decimal places, and `%.6f` specifies six decimal places. In Program 2-22, a value is displayed rounded to one, two, three, four, five, and six decimal places.

Program 2-22 (decimal_places.py)

```
# This program demonstrates how a value can be
# formatted, rounded to different numbers of
```

(program continues)

Program 2-22 *(continued)*

```
# decimal places.
my-value = 1.123456789
print '%.1f' % my-value # Rounded to 1 decimal place
print '%.2f' % my-value # Rounded to 2 decimal places
print '%.3f' % my-value # Rounded to 3 decimal places
print '%.4f' % my-value # Rounded to 4 decimal places
print '%.5f' % my-value # Rounded to 5 decimal places
print '%.6f' % my-value # Rounded to 6 decimal places
```

Program Output

```
1.1
1.12
1.123
1.1235
1.12346
1.123457
```

Formatting Multiple Values

The previous examples show how to format one value with the string formatting operator. You can format several values, using the following general format:

```
print string % (number, number, ...)
```

In the general format, *`string`* is a string that contains multiple formatting specifiers. (number, number, ...) is a list of variables or expressions enclosed in parentheses and separated by commas. The first value or expression in this list will be formatted according to the first formatting specifier in *`string`*, the second value or expression will be formatted according to the second formatting specifier in *`string`*, and so forth. Here is an example:

```
value1 = 6.7891234
value2 = 1.2345678
print 'The values are %.1f and %.3f' % (valuel, value2)
```

In the `print` statement, the `%.1f` formatting specifier corresponds to the `value1` variable and the `%.3f` formatting specifier corresponds to the `value2` variable. When the code runs, it will produce the following output:

```
The values are 6.8 and 1.235
```

NOTE: You must have the same number of formatting specifiers as values to be formatted. Otherwise an error will occur.

Specifying a Minimum Field Width

A formatting specifier can also include a minimum field width, which is the minimum number of spaces that should be used to display the value.

```
my_value = 1.123456789
print 'The value is:%6.2f' % my-value
```

In the `print` statement, the formatting specifier is `%6.2f`. The 6 specifies that the number of spaces reserved on the screen for the value should be a minimum of 6. The output of the statement will be

```
The value is:  1.12
```

In this case, the number that is displayed is shorter than the field that it is displayed in. The number 1.12 uses only four spaces on the screen, but it is displayed in a field that is six spaces wide. When this is the case, the number will be right justified in the field. If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it.

Field widths can help when you need to print values aligned in columns. For example, look at Program 2-23. Each of the variables is displayed in a field that is seven spaces wide,

Program'2-23 (columns.py)

```
# This program displays the following
# floating-point numbers in a column
# with their decimal points aligned.
numl = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.821
num5 = 88.081
num6 = 799.999

# Display each number in a field of 7 spaces
# with 2 decimal places.
print '%7.2f' % numl
print '%7.2f' % num2
print '%7.2f' % num3
print '%7.2f' % num4
print '%7.2f' % num5
print '%7.2f' % num6
```

Program Output

```
 127.90
3465.15
   3.78
 264.82
  88.08
 800.00
```

Formatting Integers and Strings

In addition to floating-point values, Python provides formatting specifiers for integers and strings. For example, the following code shows how to use the `%d` formatting specifier to format an integer:

```
hours = 40
print 'I worked %d hours this week.' % hours
```

This code will display the following:

```
I worked 40 hours this week.
```

Here is an example that formats two integer values:

```
dogs = 2
cats = 3
print 'We have %d dogs and %d cats.' % (dogs, cats)
```

This code will display

```
We have 2 dogs and 3 cats.
```

Here is an example of how the `%s` formatting specifier can be used to format a string:

```
name = 'Ringo'
print 'Hello %s. Good to see you!' % name
```

This code will display the following:

```
Hello Ringo. Good to see you!
```

The following example shows how a string and a floating-point number can be formatted in the same statement:

```
day = 'Monday'
sales = 8450.55
print 'The sales on %s were $%.2f.' % (day, sales)
```

The output is

```
The sales on Monday were $8450.55.
```

You can also apply minimum field widths to the `%d` and `%s` formatting specifiers. For example, Program 2-24 prints a series of salesperson names and units sold in two columns. Each column uses a field width of 15 spaces.

Program 2-24 (names_and_sales.py)

```
# This program displays a set of salesperson
# names and units sold in two columns.

# Assign the names to variables.
salesperson1 = 'Graves'
salesperson2 = 'Harrison'
salesperson3 = 'Hoyle'
salesperson4 = 'Kramer'
salesperson5 = 'Smith'

# Assign the units sold to variables.
units1 = 1456.78
units2 = 2890.55
units3 = 946.77
units4 = 2678.91
```

```
units5 = 1287.87

# Display the data.
print '%15s %15s' % ('Salesperson',  'Units Sold')
print '%15s %15d' % (salespersonl, unitsl)
print '%15s %15d' % (salesperson2, units2)
print '%15s %15d' % (salesperson3, units3)
print '%15s %15d' % (salesperson4, units4)
print '%15s %15d' % (salesperson5, units5)
```

Program Output

```
Salespersc[illegible]        Units  Sold
Gra[illegible]               1456
Ha[illegible]                2890
Hoy[illegible]               946
Kramer                       2678
Smith                        1287
```

Review Questions

Multiple Choice

1. A __________ error does not prevent the program from running, but causes it to produce incorrect results.
   a. syntax
   b. hardware
   c. logic
   d. fatal
2. A __________ is a single function that the program must perform in order to satisfy the customer.
   a. task
   b. software requirement
   c. prerequisite
   d. predicate
3. A(n) __________ is a set of well-defined logical steps that must be taken to perform a task.
   a. logarithm
   b. plan of action
   c. logic schedule
   d. algorithm
4. An informal language that has no syntax rules, and is not meant to be compiled or executed is called __________.
   a. faux code
   b. pseudocode
   c. Python
   d. a flowchart

5. A ________ is a diagram that graphically depicts the steps that take place in a program.
   a. flowchart
   b. step chart
   c. code graph
   d. program graph
6. A ________ is a sequence of characters.
   a. char sequence
   b. character collection
   c. string
   d. text block
7. A ________ is a name that references a value in the computer's memory.
   a. variable
   b. register
   c. RAM slot
   d. byte
8. A ________ is any hypothetical person using a program and providing input for it.
   a. designer
   b. user
   c. guinea pig
   d. test subject
9. A string literal in Python must be enclosed in
   a. parentheses
   b. single-quotes
   c. double-quotes
   d. either single-quotes or double-quotes
10. Short notes placed in different parts of a program explaining how those parts of the program work are called ________.
    a. comments
    b. reference manuals
    c. tutorials
    d. external documentation
11. A(n) ________ makes a variable reference a value in the computer's memory.
    a. variable declaration
    b. assignment statement
    c. math expression
    d. string literal
12. This symbol marks the beginning of a comment in Python.
    a. &
    b. *
    c. **
    d. #

13. Which of the following statements will cause an error?
    a. `x = 17`
    b. `17 = x`
    c. `x = 99999`
    d. `x = '17'`
14. In the expression 12 + 7, the values on the right and left of the + symbol are called ________.
    a. operands
    b. operators
    c. arguments
    d. math expressions
15. This is an operator that raises a number to a power.
    a. `%`
    b. `*`
    c. `**`
    d. `/`
16. This operator performs division, but instead of returning the quotient it returns the remainder.
    a. `%`
    b. `*`
    c. `**`
    d. `/`
17. Suppose the following statement is in a program: `price = 99.0`. After this statement executes, the price variable will reference a value of this data type.
    a. `int`
    b. `float`
    c. `currency`
    d. `str`
18. This built-in function can be used to read a number that has been typed on the keyboard.
    a. `input()`
    b. `get_num()`
    c. `read_number()`
    d. `keyboard()`
19. This built-in function can be used to convert an `int` value to a `float`.
    a. `int_to_float()`
    b. `float()`
    c. `convert()`
    d. `int()`
20. This is the string format operator.
    a. `%`
    b. `*`
    c. `&`
    d. `/`

True or False

1. Programmers must be careful not to make syntax errors when writing pseudocode programs.
2. In a math expression, multiplication and division takes place before addition and subtraction.
3. Variable names can have spaces in them.
4. In Python the first character of a variable name cannot be a number.
5. If you print a variable that has not been assigned a value, the number 0 will be displayed.

Short Answer

1. What does a professional programmer usually do first to gain an understanding of a problem?
2. What is pseudocode?
3. Computer programs typically perform what three steps?
4. If a math expression adds a `float` to an `int`, what will the data type of the result be?
5. How can the following statement be modified to make sure integer division does not take place?

```
result = a / b
```

Algorithm Workbench

1. Write Python code that prompts the user to enter his or her height and assigns the user's input to a variable named `height`.
2. Write Python code that prompts the user to enter his or her favorite color and assigns the user's input to a variable named `color`.
3. Write assignment statements that perform the following operations with the variables `a`, `b`, and `c`.
    a. Adds 2 to `a` and assigns the result to `b`
    b. Multiplies `b` times 4 and assigns the result to `a`
    c. Divides `a` by 3.14 and assigns the result to `b`
    d. Subtracts 8 from `b` and assigns the result to `a`
4. Assume the variables `result`, `w`, `x`, `y`, and `z` are all integers, and that `w = 5, x = 4, y = 8`, and `z = 2`. What value will be stored in `result` after each of the following statements execute?
    a. `result = x + y`
    b. `result = z * 2`
    c. `result = y / x`
    d. `result = y – z`
5. Write a Python statement that assigns the sum of 10 and 14 to the variable `total`.
6. Write a Python statement that subtracts the variable `down_payment` from the variable `total` and assigns the result to the variable `due`.

7. Write a Python statement that multiplies the variable `subtotal` by 0.15 and assigns the result to the variable `total`.
8. What would the following display?

   ```
   a = 5
   b = 2
   c = 3
   result = a + b * c
   print result
   ```

9. What would the following display?

   ```
   num = 99
   num = 5
   print num
   ```

10. Assume the variable `sales` references a `float` value. Write a statement that displays the value rounded to two decimal points.

Programming Exercises

1. Personal Information

Write a program that displays the following information:

- Your name
- Your address, with city, state, and ZIP
- Your telephone number
- Your college major

2. Sales Prediction

A company has determined that its annual profit is typically 23 percent of total sales. Write a program that asks the user to enter the projected amount of total sales, and then displays the profit that will be made from that amount.

Hint: use the value 0.23 to represent 23 percent.

3. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that asks the user to enter the total square feet in a tract of land and calculates the number of acres in the tract.

Hint: divide the amount entered by 43,560 to get the number of acres.

4. Total Purchase

A customer in a store is purchasing five items. Write a program that asks for the price of each item, and then displays the subtotal of the sale, the amount of sales tax, and the total. Assume the sales tax is 6 percent.

5. Distance Traveled

Assuming there are no accidents or delays, the distance that a car travels down the interstate can be calculated with the following formula:

$$Distance = Speed \times Time$$

A car is traveling at 60 miles per hour. Write a program that displays the following:

- The distance the car will travel in 5 hours
- The distance the car will travel in 8 hours
- The distance the car will travel in 12 hours

6. Sales Tax

Write a program that will ask the user to enter the amount of a purchase. The program should then compute the state and county sales tax. Assume the state sales tax is 4 percent and the county sales tax is 2 percent. The program should display the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax).

Hint: use the value 0.02 to represent 2 percent, and 0.04 to represent 4 percent.

7. Miles-per-Gallon

A car's miles-per-gallon (MPG) can be calculated with the following formula:

$$MPG = \text{Miles driven} / \text{Gallons of gas used}$$

Write a program that asks the user for the number of miles driven and the gallons of gas used. It should calculate the car's miles-per-gallon and display the result.

8. Tip, Tax, and Total

Write a program that calculates the total amount of a meal purchased at a restaurant. The program should ask the user to enter the charge for the food, and then calculate the amount of a 15 percent tip and 7 percent sales tax. Display each of these amounts and the total.

9. Celsius to Fahrenheit Temperature Converter

Write a program that converts Celsius temperatures to Fahrenheit temperatures. The formula is as follows:

$$F = \frac{9}{5}C + 32$$

The program should ask the user to enter a temperature in Celsius, and then display the temperature converted to Fahrenheit.

10. Stock Transaction Program

Last month Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:

- The number of shares that Joe purchased was 1,000.
- When Joe purchased the stock, he paid $32.87 per share.
- Joe paid his stockbroker a commission that amounted to 2 percent of the amount he paid for the stock.

Two weeks later Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 1,000.
- He sold the stock for $33.92 per share.
- He paid his stockbroker another commission that amounted to 2 percent of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount that Joe sold the stock for.
- The amount of commission Joe paid his broker when he sold the stock.
- Display the amount of money that Joe had left when he sold the stock and paid his broker (both times). If this amount is positive, then Joe made a profit. If the amount is negative, then Joe lost money.

CHAPTER

3 Simple Functions

TOPICS

3.1 Introduction to Functions
3.2 Defining and Calling a Function
3.3 Designing a Program to Use Functions
3.4 Local Variables
3.5 Passing Arguments to Functions
3.6 Global Variables and Global Constants

3.1 Introduction to Functions

CONCEPT: A function is a group of statements that exist within a program for the purpose of performing a specific task.

In Chapter 2 we described a simple algorithm for calculating an employee's pay. In the algorithm, the number of hours worked is multiplied by an hourly pay rate. A more realistic payroll algorithm, however, would do much more than this. In a real-world application, the overall task of calculating an employee's pay would consist of several subtasks, such as the following:

- Getting the employee's hourly pay rate
- Getting the number of hours worked
- Calculating the employee's gross pay
- Calculating overtime pay
- Calculating withholdings for taxes and benefits
- Calculating the net pay
- Printing the paycheck

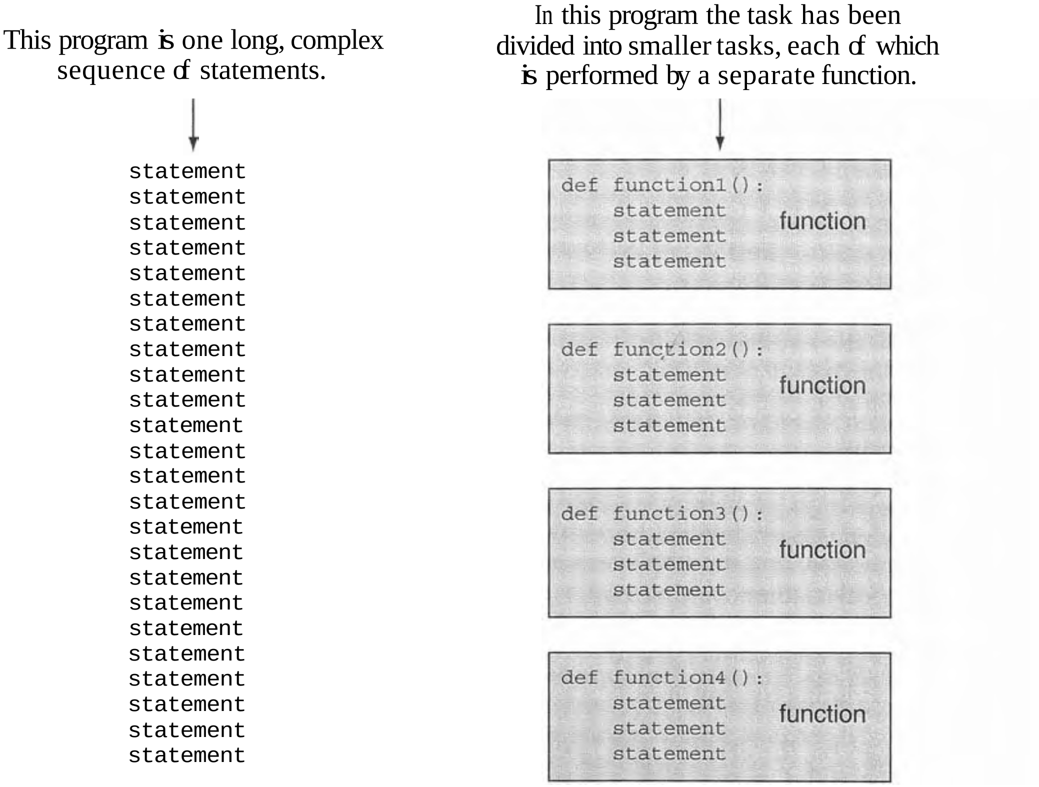
Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into small manageable pieces known as functions. A function is a group of statements that exist within a program for the purpose of performing a specific task. Instead of writing a large program as one long sequence of statements, it can be written as several small functions, each one performing a specific part of the task. These small functions can then be executed in the desired order to perform the overall task.

This approach is sometimes called *divide and conquer* because a large task is divided into several smaller tasks that are easily performed. Figure 3-1 illustrates this idea by comparing two programs: one that uses a long complex sequence of statements to perform a task, and another that divides a task into smaller tasks, each of which is performed by a separate function.

When using functions in a program, you generally isolate each task within the program in its own function. For example, a realistic pay calculating program might have the following functions:

- A function that gets the employee's hourly pay rate
- A function that gets the number of hours worked
- A function that calculates the employee's gross pay
- A function that calculates the overtime pay
- A function that calculates the withholdings for taxes and benefits
- A function that calculates the net pay
- A function that prints the paycheck

Figure 3-1 Using functions to divide and conquer a large task


This program is one long, complex sequence of statements.
In this program the task has been divided into smaller tasks, each of which is performed by a separate function.
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
def function1():
statement
statement
statement
function
def function2():
statement
statement
statement
function
def function3():
statement
statement
statement
function
def function4():
statement
statement
statement
function

Benefits of Using Functions

A program benefits in the following ways when it is broken down into functions:

Simpler Code

A program's code tends to be simpler and easier to understand when it is broken down into functions. Several small functions are much easier to read than one long sequence of statements.

Code Reuse

Functions also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform that operation, and then be executed any time it is needed. This benefit of using functions is known as code reuse because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

Better Testing

When each task within a program is contained in its own function, testing and debugging becomes simpler. Programmers can test each function in a program individually, to determine whether it correctly performs its operation. This makes it easier to isolate and fix errors.

Faster Development

Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs perform several common tasks, such as asking for a username and a password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks multiple times. Instead, functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.

Easier Facilitation of Teamwork

Functions also make it easier for programmers to work in teams. When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Checkpoint

3.1 What is a function?

3.2 What is meant by the phrase "divide and conquer?"

3.3 How do functions help you reuse code in a program?

3.4 How can functions make the development of multiple programs faster?

3.5 How can functions make it easier for programs to be developed by teams of programmers?

3.2 Defining and Calling a Function

CONCEPT: The code for a function is known as a function definition. To execute the function, you write a statement that calls it.

Function Names

Before we discuss the process of creating and using functions, we should mention a few things about function names. Just as you name the variables that you use in a program, you also name the functions. A function's name should be descriptive enough so that anyone reading your code can reasonably guess what the function does.

Python requires that you follow the same rules that you follow when naming variables, which we recap here:

- You cannot use one of Python's key words as a function name. (See Table 1-2 for a list of the key words.)
- A function name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct.

Because functions perform actions, most programmers prefer to use verbs in function names. For example, a function that calculates gross pay might be named `calculate_gross_pay`. This name would make it evident to anyone reading the code that the function calculates something. What does it calculate? The gross pay, of course. Other examples of good function names would be `get_hours`, `get_pay_rate`, `calculate_overtime`, `print_check`, and so on. Each function name describes what the function does.

Defining and Calling a Function

To create a function you write its definition. Here is the general format of a function definition in Python:

```
def function_name():
    statement
    statement
    etc.
```

The first line is known as the function header. It marks the beginning of the function definition. The function header begins with the key word `def`, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Beginning at the next line is a set of statements known as a block. A block is simply a set of statements that belong together as a group. These statements are performed any time the function is executed. Notice in the general format that all of the statements in the block are indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

Let's look at an example of a function. Keep in mind that this is not a complete program. We will show the entire program in a moment.

```
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'
```

This code defines a function named **`message`**. The **`message`** function contains a block with two `print` statements. Executing the function will cause these `print` statements to execute.

Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must call it. This is how we would call the **`message`** function:

```
message()
```

When a function is called, the interpreter jumps to that function and executes the statements in its block. Then, when the end of the block is reached, the interpreter jumps back to the part of the program that called the function, and the program resumes execution at that point. When this happens, we say that the function returns. To fully demonstrate how function calling works, we will look at Program 3-1.

Program 3-1 (function_demo.py)

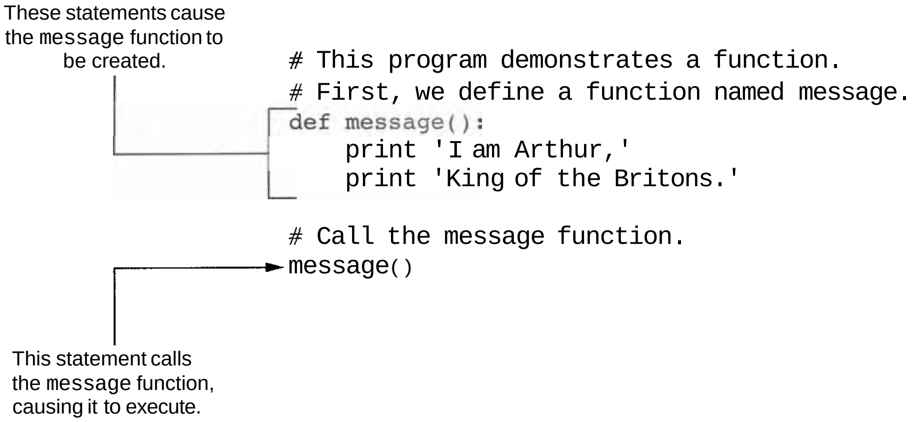
```
1   # This program demonstrates a function.
2   #First, we define a function namedmessage.
3   def message():
4       print 'I am Arthur,'
5       print 'King of the Britons.'
6
7   # Call the message function.
8   message()
```

Program Output

```
I am Arthur,
King of the Britons.
```

Let's step through this program and examine what happens when it runs. First, the interpreter ignores the comments that appear in lines 1 and 2. Then, it reads the **`def`** statement in line 3. This causes a function named **`message`** to be created in memory, containing the block of statements in lines 4 and 5. (Remember, a function definition creates a function, but it does not cause the function to execute.) Next, the interpreter encounters the comment in line 7, which is ignored. Then it executes the statement in line 8, which is a function call. This causes the **`message`** function to execute, which prints the two lines of output. Figure 3-2 illustrates the parts of this program.

Figure 3-2 The function definition and the function call


These statements cause the message function to be created.
This statement calls the message function, causing it to execute.

```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'

# Call the message function.
message()
```

Program 3-1 has only one function, but it is possible to define many functions in a program. In fact, it is common for a program to have a `main` function that is called when the program starts. The `main` function then calls other functions in the program as they are needed. It is often said that the `main` function contains a program's ***mainline logic***, which is the overall logic of the program. Program 3-2 shows an example of a program with two functions: `main` and `message`.

Program 3-2 (two_functions.py)

```
 1  # This program has two functions. First we
 2  # define the main function.
 3  def main():
 4      print 'I have a message for you.'
 5      message()
 6      print 'Goodbye!'
 7
 8  # Next we define the message function.
 9  def message():
10      print 'I am Arthur,'
11      print 'King of the Britons.'
12
13  # Call the main function.
14  main()
```

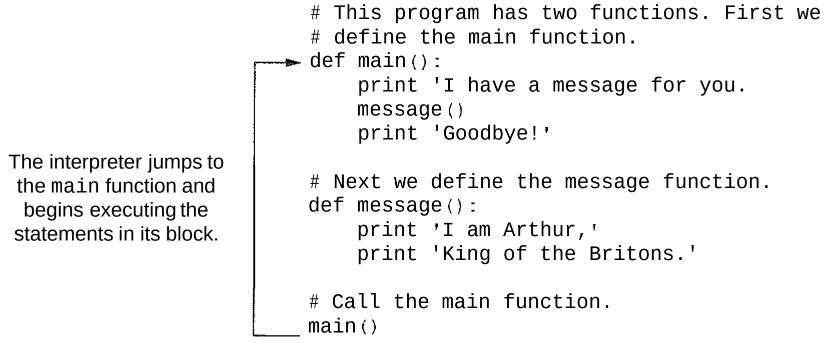
Program Output

```
I have a message for you.
I am Arthur,
King of the Britons.
Goodbye!
```

The definition of the `main` function appears in lines 3 through 6, and the definition of the `message` function appears in lines 9 through 11. The statement in line 14 calls the `main` function, as shown in Figure 3-3

The first statement in the `main` function is the `print` statement in line 4. It displays the string `'I have a message for you'`.Then, the statement in line 5 calls the `message` function. This causes the interpreter to jump to the `message` function, as shown in Figure 3-4 After the statements in the `message` function have executed, the interpreter returns to the `main` function and resumes with the statement that immediately follows the function call. As shown in Figure 3-5, this is the `print` statement that displays the string `'Goodbye!'`.

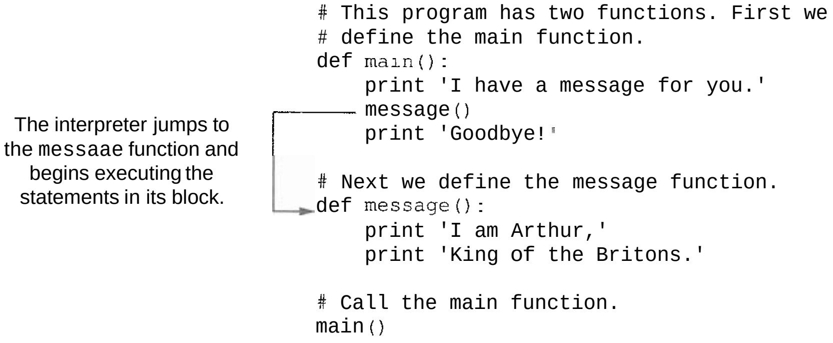
Figure 3-3 Calking the `main` function



The interpreter jumps to the `main` function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print 'I have a message for you.
    message()
    print 'Goodbye!'

# Next we define the message function.
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'

# Call the main function.
main()
```

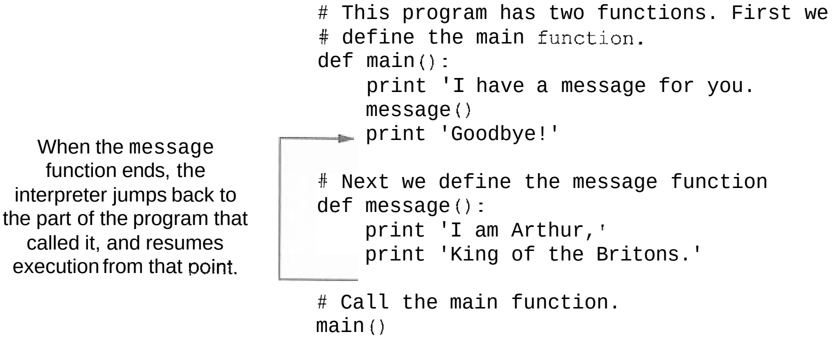
Figure 3-4 Calling the `message` function



The interpreter jumps to the `messaae` function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print 'I have a message for you.'
    message()
    print 'Goodbye!'

# Next we define the message function.
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'

# Call the main function.
main()
```

Figure 3-5 The `message` function returns

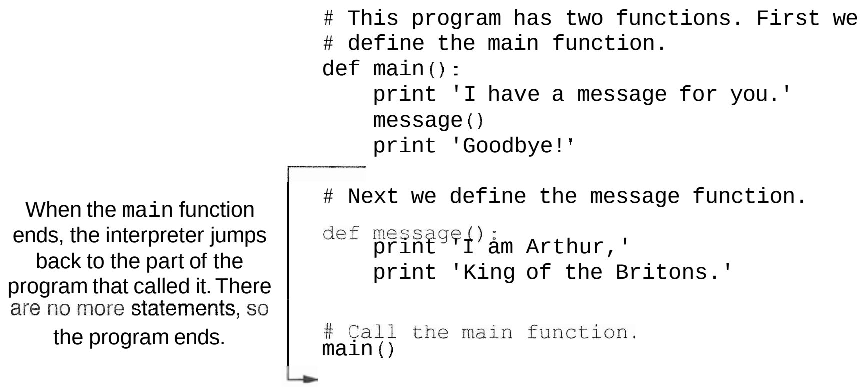


When the `message` function ends, the interpreter jumps back to the part of the program that called it, and resumes execution from that point.

```
# This program has two functions. First we
# define the main function.
def main():
    print 'I have a message for you.
    message()
    print 'Goodbye!'

# Next we define the message function
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'

# Call the main function.
main()
```

That is the end of the `main` function, so the function returns as shown in Figure 3-6. There are no more statements to execute, so the program ends.

Figure 3-6 The `main` function returns



```
# This program has two functions. First we
# define the main function.
def main():
    print 'I have a message for you.'
    message()
    print 'Goodbye!'

# Next we define the message function.
def message():
    print 'I am Arthur,'
    print 'King of the Britons.'

# Call the main function.
main()
```

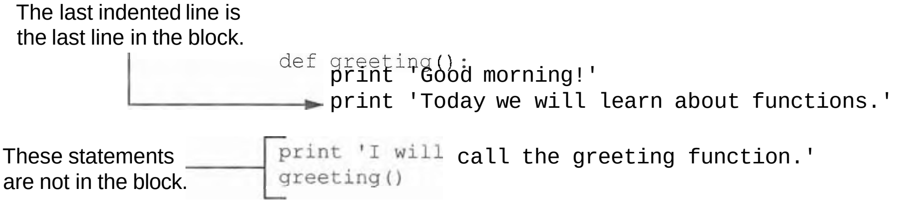
When the main function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

> NOTE: When a program calls a function, programmers commonly say that the control of the program transfers to that function. This simply means that the function takes control of the program's execution.

Indentation in Python

In Python, each line in a block must be indented. As shown in Figure 3-7, the last indented line after a function header is the last line in the function's block.

Figure 3-7 All of the statements in a block are indented



```
def greeting():
    print 'Good morning!'
    print 'Today we will learn about functions.'

print 'I will call the greeting function.'
greeting()
```

The last indented line is the last line in the block.

These statements are not in the block.

When you indent the lines in a block, make sure each line begins with the same number of spaces. Otherwise an error will occur. For example, the following function definition will cause an error because the lines are all indented with different numbers of spaces.

```
def my-function():
  print 'And now for'
print 'something completely'
   print 'different.'
```

In an editor there are two ways to indent a line: (1) by pressing the Tab key at the beginning of the line, or (2) by using the spacebar to insert spaces at the beginning of the line. You can use either tabs or spaces when indenting the lines in a block, but don't use both. Doing so may confuse the Python interpreter and cause an error.

IDLE, as well as most other Python editors, automatically indents the lines in a block. When you type the colon at the end of a function header, all of the lines typed afterward will automatically be indented. After you have typed the last line of the block you press the Backspace key to get out of the automatic indentation.

TIP: Python programmers customarily use four spaces to indent the lines in a block. You can use any number of spaces you wish, as long as all the lines in the block are indented by the same amount.

NOTE: Blank lines that appear in a block are ignored.

Checkpoint

3.6 A function definition has what two parts?

3.7 What does the phrase "calling a function" mean?

3.8 When a function is executing, what happens when the end of the function's block is reached?

3.9 Why must you indent the statements in a block?

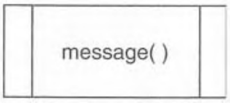
3.3 Designing a Program to Use Functions

CONCEPT: Programmers commonly use a technique known as top-down design to break down an algorithm into functions.

Flowcharting a Program with Functions

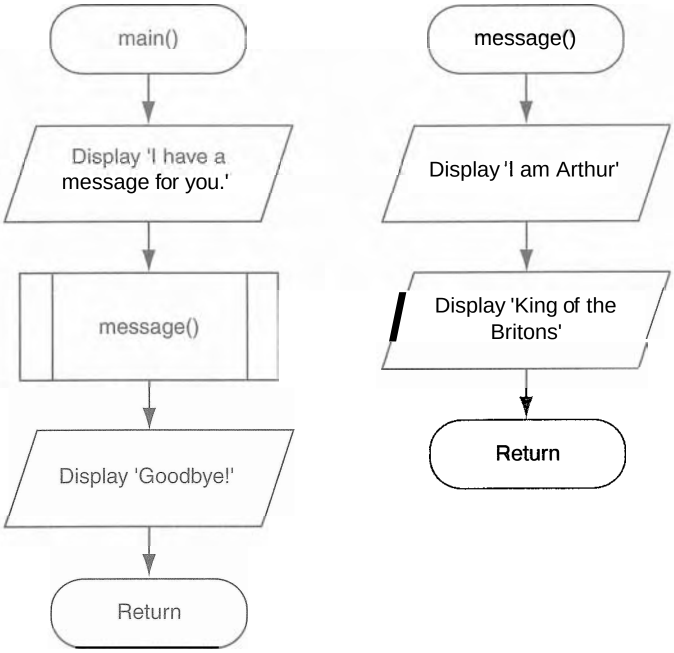
In Chapter 2 we introduced flowcharts as a tool for designing programs. In a flowchart, a function call is shown with a rectangle that has vertical bars at each side, as shown in Figure *3-8*. The name of the function that is being called is written on the symbol. The example shown in Figure *2-8* shows how we would represent a call to the `message` function.

Figure 3-8 Function call symbol



Programmers typically draw a separate flowchart for each function in a program. For example, Figure 3-9 shows how the `main` function and the `message` function in Program 3-2 would be flowcharted. When drawing a flowchart for a function, the starting terminal symbol usually shows the name of the function and the ending terminal symbol usually reads `Return`.

Figure 3-9 Flowchart for Program 3-2



Top-Down Design

In this section, we have discussed and demonstrated how functions work. You've seen how control of a program is transferred to a function when it is called, and then returns to the part of the program that called the function when the function ends. It is important that you understand these mechanical aspects of functions.

Just as important as understanding how functions work is understanding how to design a program that uses functions. Programmers commonly use a technique known as *top-down design* to break down an algorithm into functions. The process of top-down design is performed in the following manner:

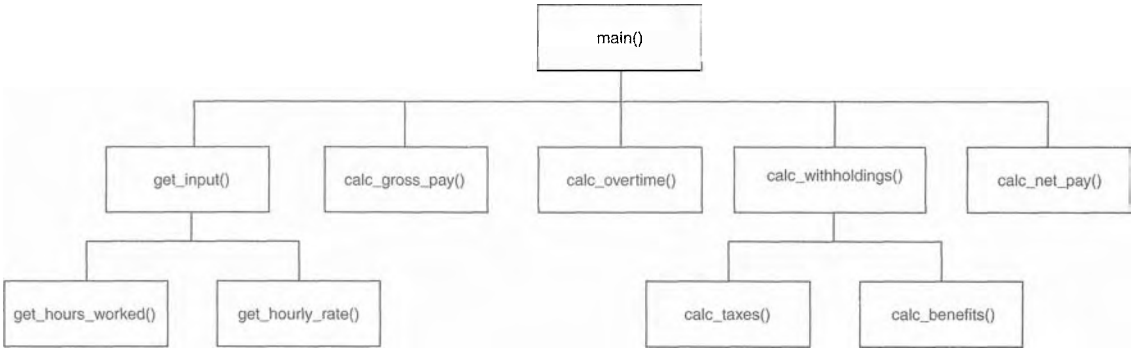
- The overall task that the program is to perform is broken down into a series of subtasks.
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.
- Once all of the subtasks have been identified, they are written in code.

This process is called top-down design because the programmer begins by looking at the topmost level of tasks that must be performed, and then breaks down those tasks into lower levels of subtasks.

Hierarchy Charts

Flowcharts are good tools for graphically depicting the flow of logic inside a function, but they do not give a visual representation of the relationships between functions. Programmers commonly use *hierarchy charts* for this purpose. A hierarchy chart, which is also known as a *structure chart*, shows boxes that represent each function in a program. The boxes are connected in a way that illustrates the functions called by each function. Figure 3-10 shows an example of a hierarchy chart for a hypothetical pay calculating program.

Figure 3-10 A hierarchy chart



The chart shown in Figure 3-9 shows the `main` function as the topmost function in the hierarchy. The main function calls five other functions: `get_input`, `calc_gross_pay`, `calc_overtime`, `calc_withholdings`, and `calc_net_pay`. The `get_input` function calls two additional functions: `get_hours_worked` and `get_hourly_rate`. The `calc_withholdings` function also calls two functions: `calc_taxes` and `calc_benefits`.

Notice that the hierarchy chart does not show the steps that are taken inside a function. Because they do not reveal any details about how functions work, they do not replace flowcharts or pseudocode.

In the Spotlight:

Defining and Calling Functions

Professional Appliance Service, Inc. offers maintenance and repair services for household appliances. The owner wants to give each of the company's service technicians a small handheld computer that displays step-by-step instructions for many of the repairs that they perform. To see how this might work, the owner has asked you to develop a program that displays the following instructions for disassembling an Acme laundry dryer:

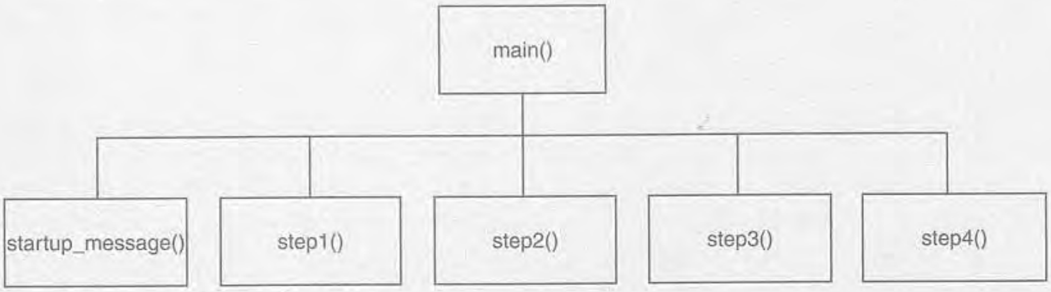
Step 1: Unplug the dryer and move it away from the wall.
Step 2: Remove the six screws from the back of the dryer.
Step 3: Remove the dryer's back panel.
Step 4: Pull the top of the dryer straight up.

During your interview with the owner, you determine that the program should display the steps one at a time. You decide that after each step is displayed, the user will be asked to press the Enter key to see the next step. Here is the algorithm in pseudocode:

```
Display a starting message, explaining what the program does.
Ask the user to press Enter to see step 1.
Display the instructions for step 1.
Ask the user to press Enter to see the next step.
Display the instructions for step 2.
Ask the user to press Enter to see the next step.
Display the instructions for step 3.
Ask the user to press Enter to see the next step.
Display the instructions for step 4.
```

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the program's `main` function. Figure 3-11 shows the program's structure in a hierarchy chart.

Figure 3-11 Hierarchy chart for the program



As you can see from the hierarchy chart, the `main` function will call several other functions. Here are summaries of those functions:

- `startup_message`—This function will display the starting message that tells the technician what the program does.
- `step1`—This function will display the instructions for step 1.
- `step2`—This function will display the instructions for step 2.
- `step3`—This function will display the instructions for step 3.
- `step4`—This function will display the instructions for step 4.

Between calls to these functions, the `main` function will instruct the user to press a key to see the next step in the instructions. Program 3-3 shows the code for the program.

Program 3-3 (acme_dryer.py)

```
# This program displays step-by-step instructions
# for disassembling an Acme dryer.
# The main function performs the program's main logic.
def main():
    # Display the start-up message.
    startup_message()
    raw_input('Press Enter to see Step 1.')
    # Display step 1.
    step1()
    raw_input('Press Enter to see Step 2.')
    # Display step 2.
    step2()
    raw_input('Press Enter to see Step 3.')
    # Display step 3.
    step3()
    raw_input('Press Enter to see Step 4.')
    # Display step 4.
    step4()

# The startup_message function displays the
# program's initial message on the screen.
def startup_message():
    print 'This program tells you how to'
    print 'disassemble an ACME laundry dryer.'
    print 'There are 4 steps in the process.'
    print                  # Print a blank line.

# The step1 function displays the instructions
# for step 1.
def step1():
    print 'Step 1: Unplug the dryer and'
    print 'move it away from the wall.'
    print                  # Print a blank line.

# The step2 function displays the instructions
# for step 2.
def step2():
    print 'Step 2: Remove the six screws'
    print 'from the back of the dryer.'
    print                  # Print a blank line.

# The step3 function displays the instructions
# for step 3.
def step3():
    print 'Step 3: Remove the back panel'
```

(program continues)

Program 3-3 *(continued)*

```
46        print 'from the dryer.'
47        print                    # Print a blank line.
48
49    # The step4 function displays the instructions
50    # for step 4.
51    def step4():
52        print 'Step 4: Pull the top of the'
53        print 'dryer straight up.'
54
55    # Call the main function to begin the program.
56    main()
```

Program Output (with input shown in bold)

```
This program tells you how to
disassemble an ACME laundry dryer.
There are 4 steps in the process.

Press Enter to see Step 1. [Enter]
Step 1: Unplug the dryer and
move it away from the wall.

Press Enter to see Step 2. [Enter]
Step 2: Remove the six screws
from the back of the dryer.

Press Enter to see Step 3. [Enter]
Step 3: Remove the back panel
from the dryer.

Press Enter to see Step 4. [Enter]
Step 4: Pull the top of the
dryer straight up.
```

Pausing Execution Until the User Presses Enter

Sometimes you want a program to pause so the user can read information that has been displayed on the screen. When the user is ready for the program to continue execution, he or she presses the Enter key and the program resumes. In Python you can use the `raw—input` function to cause a program to pause until the user presses the Enter key. Line 7 in Program 3-3 is an example:

```
raw-input('Press Enter to see Step 1.')
```

This statement displays the prompt `'Press Enter to see Step 1.'` and pauses until the user to presses the Enter key. The program also uses this technique in lines 10, 13, and 16.

3.4 Local Variables

CONCEPT: **A** local variable is created inside a function and cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

Anytime you assign a value to a variable inside a function, you create a *local variable.* A local variable belongs to the function in which it is created, and only statements inside that function can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the function in which it is created.)

An error will occur if a statement in one function tries to access a local variable that belongs to another function. For example, look at Program 3-4.

Program 3-4 (bad_local.py)

```
 1  # Definition of the main function.
 2  def main():
 3      get-name()
 4      print 'Hello', name       # This causes an error!
 5
 6  # Definition of the get-name function.
 7  def get-name():
 8      name = raw-input('Enter your name: ')
 3
10  # Call the main function.
11  main()
```

This program has two functions: `main` and `get` name. In line 8 the name variable is assigned a value that is entered by the user. This statement is inside the `get_name` function, so the name variable is local to that function. This means that the name variable cannot be accessed by statements outside the get—name function.

The `main` function calls the get—name function in line **3.** Then, the `print` statement in line 4 tries to access the name variable. This results in an error because the name variable is local to the get—name function, and statements in the `main` function cannot access it.

Scope and Local Variables

A variable's *scope* is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope. A local variable's scope is the function in which the variable is created. As you saw demonstrated in Program 3-4, no statement outside the function may access the variable.

In addition, a local variable cannot be accessed by code that appears inside the function at a point before the variable has been created. For example, look at the following function. It will cause an error because the `print` statement tries to access the `val` variable, but this statement appears before the `val` variable has been created. Moving the assignment statement to a line before the `print` statement will fix this error.

```
def bad-function():
    print 'The value is', val #    This will cause an error!
    val = 99
```

Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name. For example, look at the Program 3-5. In addition to the `main` function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

Program 3-5 (birds.py)

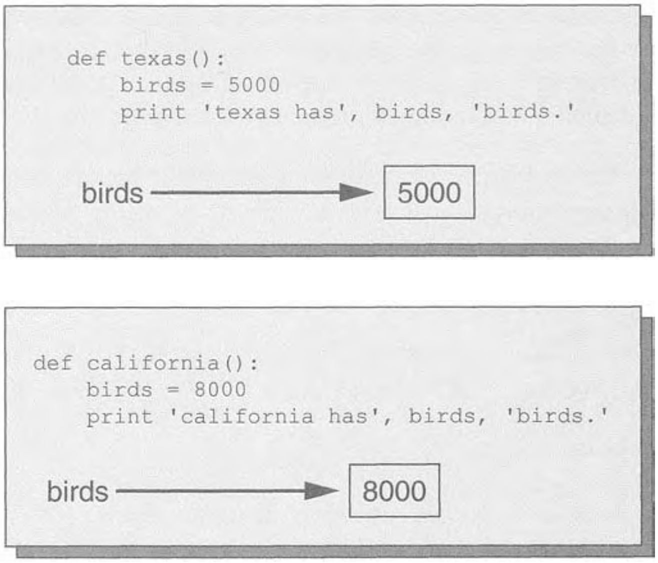
```
# This program demonstrates two functions that
# have local variables with the same name.

def main():
    # Call the texas function.
    texas()
    # Call the california function.
    california()

# Definition of the texas function. It creates
# a local variable named birds.
def texas():
    birds = 5000
    print 'texas has', birds, 'birds.'

# Definition of the california function. It also
# creates a local variable named birds.
def california():
    birds = 8000
    print 'california has', birds, 'birds.'

# Call the main function.
main()
```

Program Output

```
texas has 5000 birds.
california has 8000 birds.
```

Although there are two separate variables named birds in this program, only one of them is visible at a time because they are in different functions. This is illustrated in Figure 3-12. When the `texas` function is executing, the `birds` variable that is created in line 13 is visible. When the `california` function is executing, the `birds` variable that is created in line 19 is visible.

Figure 3-12 Each function has its own `birds` variable



```
def texas():
    birds = 5000
    print 'texas has', birds, 'birds.'
```

```
def california():
    birds = 8000
    print 'california has', birds, 'birds.'
```

Checkpoint

3.10 What is a local variable? How is access to a local variable restricted?

3.11 What is a variable's scope?

3.12 Is it permissible for a local variable in one function to have the same name as a local variable in a different function?

3.5 Passing Arguments to Functions

CONCEPT: An argument is any piece of data that is passed into a function when the function is called. A parameter is a variable that receives an argument that is passed into a function.

Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function. Pieces of data that are sent into a function are known as arguments. The function can use its arguments in calculations or other operations.

If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A *parameter variable*, often simply called a *parameter*, is a special variable that is assigned the value of an argument when a function is called. Here is an example of a function that has a parameter variable:

```
def show_double(number):
    result = number * 2
    print result
```

This function's name is `show_double`. Its purpose is to accept a number as an argument and display the value of that number doubled. Look at the function header and notice the word `number` that appear inside the parentheses. This is the name of a parameter variable. This variable will be assigned the value of an argument when the function is called. Program 3-6 demonstrates the function in a complete program.

Program 3-6 (pass–arg.py)

```
 1  # This program demonstrates an argument being
 2  # passed to a function.
 3
 4  def main():
 5      value = 5
 6      show_double(value)
 7
 8  # The show-double function accepts an argument
 9  # and displays double its value.
10  def show_double(number):
11      result = number * 2
12      print result
13
14  # Call the main function.
15  main()
```

Program Output

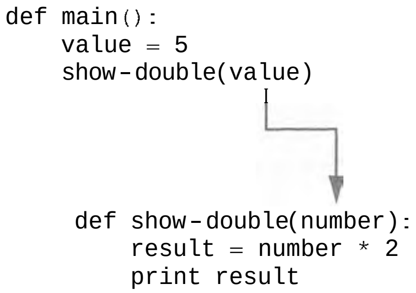
```
10
```

When this program runs, the `main` function is called in line 15. Inside the `main` function, line 5 creates a local variable named `value`, assigned the value 5. Then the following statement in line 6 calls the `show_double` function:

```
show_double(value)
```

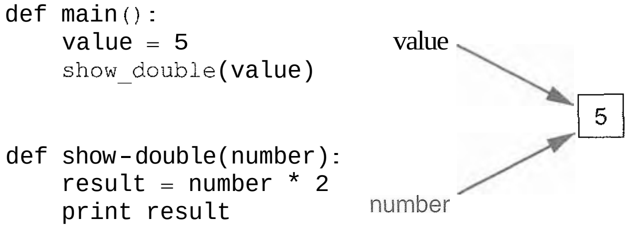
Notice that `value` appears inside the parentheses. This means that `value` is being passed as an argument to the `show_double` function, as shown in Figure 3-13 When this statement executes, the `show_double` function will be called and the `number` parameter will be assigned the same value as the `value` variable. This is shown in Figure 3-14.

Figure 3-13 The `value` variable is passed as an argument



```
def main():
    value = 5
    show-double(value)

def show-double(number):
    result = number * 2
    print result
```

Figure 3-14 The `value` variable and the `number` parameter reference the same value



```
def main():
    value = 5
    show_double(value)

def show-double(number):
    result = number * 2
    print result
```

Let's step through the `show_double` function. As we do, remember that the number parameter variable will be assigned the value that was passed to it as an argument. In this program, that number is 5.

Line 11 assigns the value of the expression `number * 2` to a local variable named `result`. Because `number` references the value 5, this statement assigns 10 to `result`. Line 12 displays the `result` variable.

The following statement shows how the `show_double` function can be called with a numeric literal passed as an argument:

```
show_double(50)
```

This statement executes the `show_double` function, assigning 50 to the `number` parameter. The function will print 100.

Parameter Variable Scope

Earlier in this chapter, you learned that a variable's scope is the part of the program in which the variable may be accessed. A variable is visible only to statements inside the variable's scope. A parameter variable's scope is the function in which the parameter is used. All of the statements inside the function can access the parameter variable, but no statement outside the function can access it.

In the Spotlight:

Passing an Argument to a Function

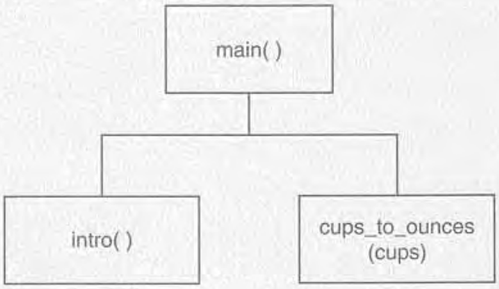
Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Display an introductory screen that explains what the program does.*
2. *Get the number of cups.*
3. *Convert the number of cups to fluid ounces and display the result.*

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the program's `main` function. Figure 3-15 shows the program's structure in a hierarchy chart.

Figure 3-15 Hierarchy chart for the program



As shown in the hierarchy chart, the `main` function will call two other functions. Here are summaries of those functions:

- `intro`—This function will display a message on the screen that explains what the program does.
- `cups_to_ounces`—This function will accept the number of cups as an argument and calculate and display the equivalent number of fluid ounces.

In addition to calling these functions, the `main` function will ask the user to enter the number of cups. This value will be passed to the `cups_to_ounces` function. The code for the program is shown in Program 3-7.

Program 3-7 (cups_to_ounces.py)

```
# This program converts cups to fluid ounces.

def main():
    # display the intro screen.
    intro()
    # Get the number of cups.
    cups_needed = input('Enter the number of cups: ')
    # Convert the cups to ounces.
    cups_to_ounces(cups_needed)

# The intro function displays an introductory screen.
def intro():
    print 'This program converts measurements'
    print 'in cups to fluid ounces. For your'
    print 'reference the formula is:'
    print ' 1 cup = 8 fluid ounces'
    print

# The cups_to_ounces function accepts a number of
# cups and displays the equivalent number of ounces.
def cups_to_ounces(cups):
    ounces = cups * 8
    print 'That converts to', ounces, 'ounces.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
This program converts measurements
in cups to fluid ounces. For your
reference the formula is:
    1 cup = 8 fluid ounces

Enter the number of cups: 4 [Enter]
That converts to 32 ounces.
```

Passing Multiple Arguments

Often it's useful to write functions that can accept multiple arguments. Program 3-8 shows a function named show—sum, that accepts two arguments. The function adds the two arguments and displays their sum.

Program 3-8 (multiple_args.py)

```
# This program demonstrates a function that accepts
# two arguments.

def main():
    print 'The sum of 12 and 45 is'
```

(program continues)

Program 3-8 *(continued)*

```
 6      show_sum(12, 45)
 7
 8  # The show-sum function accepts two arguments
 9  # and displays their sum.
10  def show_sum(num1, num2):
11      result = numl + num2
12      print result
13
14  # Call the main function.
15  main()
```

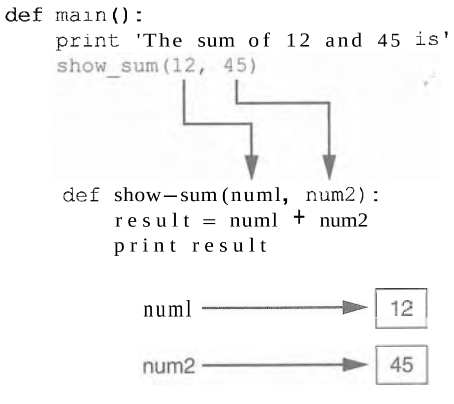
Program Output

```
The sum of 12 and 45 is
57
```

Notice that two parameter variable names, `num1` and `num2`, appear inside the parentheses in the `show_sum` function header. This is often referred to as a *parameter list.* Also notice that a comma separates the variable names.

The statement in line 6 calls the `show_sum` function and passes two arguments: 12 and 45. These arguments are passed by *position* to the corresponding parameter variables in the function. In other words, the first argument is passed to the first parameter variable, and the second argument is passed to the second parameter variable. So, this statement causes 12 to be assigned to the `num1` parameter and 45 to be assigned to the `num2` parameter, as shown in Figure 3-16.

Figure 3-16 Two arguments passed to two parameters



```
def main():
    print 'The sum of 12 and 45 is'
    show_sum(12, 45)

def show-sum(numl, num2):
    result = numl + num2
    print result
```

Suppose we were to reverse the order in which the arguments are listed in the function call, as shown here:

```
show_sum(45, 12)
```

This would cause 45 to be passed to the `num1` parameter and 12 to be passed to the `num2` parameter. The following code shows another example. This time we are passing variables as arguments.

```
value1 = 2
value2 = 3
show_sum(value1, value2)
```

When the show—sum function executes as a result of this code, the `num1` parameter will be assigned the value 2 and the `num2` parameter will be assigned the value 3.

Program 3-9 shows one more example. This program passes two strings as arguments to a function.

Program 3-9 (string_args.py)

```
# This program demonstrates passing two string
# arguments to a function.

def main():
    first_name = raw-input('Enter your first name: ')
    last-name = raw_input('Enter your last name: ')
    print 'Your name reversed is'
    reverse_name(first_name, last-name)

def reverse_name(first, last):
    print last, first

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your first name: Matt [Enter]
En                                oyle [Enter]
Your name reversed is
Hoyle Matt
```

Making Changes to Parameters

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value. However, any changes that are made to the parameter variable will not affect the argument. To demonstrate this look at Program 3-10.

Program 3-10 (change_me.py)

```
# This program demonstrates what happens when you
# change the value of a parameter.

```

(program continues)

Program 3-10 *(continued)*

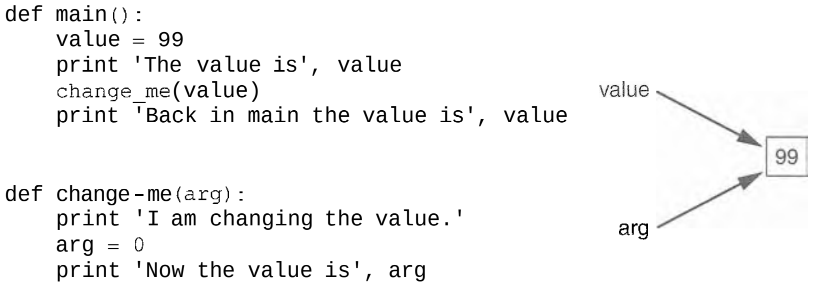
```
 4  def main():
 5      value = 99
 6      print 'The value is', value
 7      change_me(value)
 8      print 'Back in main the value is', value
 9
10  def change_me(arg):
11      print 'I am changing the value.'
12      arg = 0
13      print 'Now the value is', arg
14
15  # Call the main function.
16  main()
```

Program Output

```
The value is 99
I am changing the value.
Now the vslue is 0
Back in main the value is 99
```

The `main` function creates a local variable named `value` in line 5, assigned the value 99. The `print` statement in line 6 displays `'The value is 99'`. The `value` variable is then passed as an argument to the change—me function in line 7. This means that in the `change_me` function the `arg` parameter will also reference the value 99. This is shown in Figure 3-17.

Figure *3-87* The value variable is passed to the change—me function

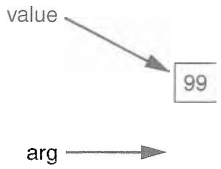


```
def main():
    value = 99
    print 'The value is', value
    change_me(value)
    print 'Back in main the value is', value


def change-me(arg):
    print 'I am changing the value.'
    arg = 0
    print 'Now the value is', arg
```

Inside the `change_me` function, in line 12, the `arg` parameter is assigned the value 0. This reassignment changes `arg`, but it does not affect the `value` variable in `main`. As shown in Figure 3-18, the two variables now reference different values in memory. The `print` statement in line 13 displays `'Now the value is 0'` and the function ends.

Figure 3-18 The `value` variable is passed to the `change_me` function

```
def main():
    value = 99
    print 'The value is', value
    change_me(value)
    print 'Back in main the value is', value


def change_me(arg):
    print 'I am changing the value.'
    arg = 0
    print 'Now the value is', arg
```



Control of the program then returns to the `main` function. The next statement to execute is the `print` statement in line 8. This statement displays `'Back in main the value is 99'`. This proves that even though the parameter variable `arg` was changed in the `change_me` function, the argument (the `value` variable in `main`) was not modified.

The form of argument passing that is used in Python, where a function cannot change the value of an argument that was passed to it, is commonly called *pass by value*. This is a way that one function can communicate with another function. The communication channel works in only one direction, however. The calling function can communicate with the called function, but the called function cannot use the argument to communicate with the calling function. In Chapter 6 you will learn how to write a function that can communicate with the part of the program that called it by returning a value.

Keyword Arguments

Programs 3-8 and 3-9 demonstrate how arguments are passed by position to parameter variables in a function. Most programming languages match function arguments and parameters this way. In addition to this conventional form of argument passing, the Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:

```
parameter_name=value
```

In this format, ***`parameter_name`*** is the name of a parameter variable and ***`value`*** is the value being passed to that parameter. An argument that is written in accordance with this syntax is known as a *keyword argument.*

Program 3-11 demonstrates keyword arguments. This program uses a function named `show_interest` that displays the amount of simple interest earned by a bank account for a number of periods. The function accepts the arguments `principal` (for the account principal), `rate` (for the interest rate per period), and `periods` (for the number of periods). When the function is called in line 7, the arguments are passed as keyword arguments.

Program 3-11 (keyword_args.py)

```
1  # This program demonstrates keyword arguments.
2
3  def main():
4      # Show the amount of simple interest, using 0.01 as
```

(program continues)

Program 3-11 *(continued)*

```
 5     # interest rate per period, 10 as the number of periods,
 6     # and $10,000 as the principal.
 7     show_interest(rate=0.01, periods=10, principal=10000.0)
 8
 9 # The show-interest function displays the amount of
10 # simple interest for a given principal, interest rate
11 # per period, and number of periods.
12
13 def show_interest(principal, rate, periods):
14     interest = principal * rate * periods
15     print 'The simple interest will be $%.2f.' % interest
16
17 # Call the main function.
18 main()
```

Program Output

```
The simple interest will be $1000.00.
```

Notice in line 7 that the order of the keyword arguments does not match the order of the parameters in the function header in line 13. Because a keyword argument specifies which parameter the argument should be passed into, its position in the function call does not matter.

Program 3-12 shows another example. This is a variation of the `string — args` program shown in Program 3-9. This version uses keyword arguments to call the `reverse — name` function.

Program 3-12 (keyword_string_args.py)

```
# This program demonstrates passing two strings as
# keyword arguments to a function.

def main():
    first-name = raw_input('Enter your first name: ')
    last-name = raw_input('Enter your last name: ')
    print 'Your name reversed is'
    reverse_name(last=last_name, first=first_name)

def reverse_name(first, last):
     print last, first

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your first name: Matt [Enter]
Enter your last name: Hoyle [Enter]
Your name reversed is
Hoyle Matt
```

Mixing Keyword Arguments with Positional Arguments

It is possible to mix positional arguments and keyword arguments in a function call, but the positional arguments must appear first, followed by the keyword arguments. Otherwise an error will occur. Here is an example of how we might call the `show_interest` function of Program 3-10 using both positional and keyword arguments:

```
show_interest(10000.0, rate=0.01, periods=10)
```

In this statement, the first argument, `10000.0`, is passed by its position to the `principal` parameter. The second and third arguments are passed as keyword arguments. The following function call will cause an error, however, because a non-keyword argument follows a keyword argument:

```
# This will cause an ERROR!
show_interest(1000.0, rate=0.01, 10)
```

Checkpoint

3.13 What are the pieces of data that are passed into a function called?

3.14 What are the variables that receive pieces of data in a function called?

3.15 What is a parameter variable's scope?

3.16 When a parameter is changed, does this affect the argument that was passed into the parameter?

3.17 The following statements call a function named `show—data`. Which of the statements passes arguments by position, and which passes keyword arguments?

```
a) show_data(name='Kathryn', age=25)
b) show_data('Kathryn', 25)
```

3.6 Global Variables and Global Constants

CONCEPT: A global variable is accessible to all the functions in a program file.

You've learned that when a variable is created by an assignment statement inside a function, the variable is local to that function. Consequently, it can be accessed only by statements inside the function that created it. When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is global. A global variable can be accessed by any statement in the program file, including the statements in any function. For example, look at Program 3-13.

Program 3-13 (global1.py)

```
# Create a global variable.
my-value = 10

# The show-value function prints
# the value of the global variable.
```

(program *continues)*

Program 3-13 *(continued)*

```
 6  def show-value():
 7      print my-value
 8
 9  # Call the show-value function.
10  show-value()
```

Program Output

```
10
```

The assignment statement in line 2 creates a variable named `my_value`. Because this statement is outside any function, it is global. When the `show_value` function executes, the statement in line 7 prints the value referenced by `my_value`.

An additional step is required if you want a statement in a function to assign a value to a global variable. In the function you must declare the global variable, as shown in Program 3-14.

Program 3-14 (global2.py)

```
 1  # Create a global variable.
 2  number = 0
 3
 4  def main():
 5      global number
 6      number = input('Enter a number: ')
 7      show-number()
 8
 9  def show-number():
10      print 'The number you entered is', number,
11
12  # Call the main function.
13  main()
```

Program Output (with input shown in bold)

```
Enter a number: [illegible] [Enter]
The number you entered is 55
```

The assignment statement in line 2 creates a global variable named number. Notice that inside the `main` function, line 5 uses the `global` key word to declare the number variable. This statement tells the interpreter that the `main` function intends to assign a value to the global number variable. That's just what happens in line 6. The value entered by the user is assigned to number.

Most programmers agree that you should restrict the use of global variables, or not use them at all. The reasons are as follows:

- Global variables make debugging difficult. Any statement in a program file can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.

- Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.

In most cases, you should create variables locally and pass them as arguments to the functions that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is permissible to use global constants in a program. A *global constant* is a global name that references a value that cannot be changed. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

Although the Python language does not allow you to create true global constants, you can simulate them with global variables. If you do not declare a global variable with the `global` key word inside a function, then you cannot change the variable's assignment inside that function. The following *In the Spotlight* section demonstrates how global variables can be used in Python to simulate global constants.

In the Spotlight:

Using Global Constants

Marilyn works for Integrated Systems, Inc., a software company that has a reputation for providing excellent fringe benefits. One of their benefits is a quarterly bonus that is paid to all employees. Another benefit is a retirement plan for each employee. The company contributes 5 percent of each employee's gross pay and bonuses to their retirement plans. Marilyn wants to write a program that will calculate the company's contribution to an employee's retirement account for a year. She wants the program to show the amount of contribution for the employee's gross pay and for the bonuses separately. Here is an algorithm for the program:

```
Get the employee's annual gross pay.
Get the amount of bonuses paid to the employee.
Calculate and display the contribution for the gross pay.
Calculate and display the contribution for the bonuses.
```

The code for the program is shown in Program 3-15.

Program 3-15 (retirement.py)

```
 1  # The following global variable represents
 2  # the contribution rate.
 3  CONTRIBUTION_RATE = 0.05
 4
 5  def main():
 6      gross_pay = input('Enter the gross pay: ')
 7      bonus = input('Enter the amount of bonuses: ')
 8      show_pay_contrib(gross_pay)
 9      show_bonus_contrib(bonus)
10
11  # The show_pay_contrib function accepts the gross
12  # pay as an argument and displays the retirement
13  # contribution for that amount of pay.
14  def show_pay_contrib(gross):
15      contrib = gross * CONTRIBUTION_RATE
16      print 'Contribution for gross pay: $%.2f' % contrib
17
18  # The show_bonus_contrib function accepts the
19  # bonus amount as an argument and displays the
20  # retirement contribution for that amount of pay.
21  def show_bonus_contrib(bonus):
22      contrib = bonus * CONTRIBUTION_RATE
23      print 'Contribution for bonuses: $%.2f' % contrib
24
25  # Call the main function.
26  main()
```

Program Output (with input shown in bold)

```
Enter the gross pay: 80000.00 [Enter]
Enter the amount of bonuses: 20000.00 [Enter]
Contribution for gross pay: $4000.00
Contribution for bonuses: $1000.00
```

First, notice the global variable that is created in line 3:

```
CONTRIBUTION_RATE = 0.05
```

This variable will be used as a global constant to represent the percentage of an employee's pay that the company will contribute to a retirement account. It is a common practice to write a constant's name in all uppercase letters. This serves as a reminder that the value referenced by the name is not to be changed in the program.

The `CONTRIBUTION_RATE` constant is used in the calculation in line 15 (in the `show_pay_contrib` function) and again in line 22 (in the `show_bonus_contrib` function).

Marilyn decided to use this global constant to represent the 5 percent contribution rate for two reasons:

- It makes the program easier to read. When you look at the calculations in lines 15 and 22 it is apparent what is happening.
- Occasionally the contribution rate changes. When this happens, it will be easy to update the program by changing the assignment statement in line 2.

Checkpoint

3.18 What is the scope of a global variable?

3.19 Give one good reason that you should not use global variables in a program.

3.20 What is a global constant? Is it permissible to use global constants in a program?

Review Questions

Multiple Choice

1. A group of statements that exist within a program for the purpose of performing a specific task is a(n) __________.
   a. block
   b. parameter
   c. function
   d. expression
2. A design technique that helps to reduce the duplication of code within a program and is a benefit of using functions is __________.
   a. code reuse
   b. divide and conquer
   c. debugging
   d. facilitation of teamwork
3. The first line of a function definition is known as the __________.
   a. body
   b. introduction
   c. initialization
   d. header
4. You __________ the function to execute it.
   a. define
   b. call
   c. import
   d. export
5. A design technique that programmers use to break down an algorithm into functions is known as __________.
   a. top-down design
   b. code simplification

   c. code refactoring
   d. hierarchical subtasking
6. A ________ is a diagram that gives a visual representation of the relationships between functions in a program.
   a. flowchart
   b. function relationship chart
   c. symbol chart
   d. hierarchy chart
7. A ________ is a variable that is created inside a function.
   a. global variable
   b. local variable
   c. hidden variable
   d. none of the above; you cannot create a variable inside a function
8. A(n) ________ is the part of a program in which a variable may be accessed.
   a. declaration space
   b. area of visibility
   c. scope
   d. mode
9. A(n) ________ is a piece of data that is sent into a function.
   a. argument
   b. parameter
   c. header
   d. packet
10. A(n) ________ is a special variable that receives a piece of data when a function is called.
    a. argument
    b. parameter
    c. header
    d. packet
11. A variable that is visible to every function in a program file is a ________.
    a. local variable
    b. universal variable
    c. program-wide variable
    d. global variable
12. When possible, you should avoid using ________ variables in a program.
    a. local
    b. global
    c. reference
    d. parameter

True or False

1. The phrase "divide and conquer" means that all of the programmers on a team should be divided and work in isolation.
2. Functions make it easier for programmers to work in teams.
3. Function names should be as short as possible.

4. Calling a function and defining a function mean the same thing.
5. A flowchart shows the hierarchical relationships between functions in a program.
6. A hierarchy chart does not show the steps that are taken inside a function.
7. A statement in one function can access a local variable in another function.
8. Most languages do not allow you to write functions that accept multiple arguments.
9. In Python, you can specify which parameter an argument should be passed into a function call.
10. You cannot have both keyword arguments and non-keyword arguments in a function call.

Short Answer

1. How do functions help you to reuse code in a program?
2. Name and describe the two parts of a function definition.
3. When a function is executing, what happens when the end of the function block is reached?
4. What is a local variable? What statements are able to access a local variable?
5. What is a local variable's scope?
6. Why do global variables make a program difficult to debug?

Algorithm Workbench

1. Write a function named `times_ten`. The function should accept an argument and display the product of its argument multiplied times 10.
2. Examine the following function header, and then write a statement that calls the function, passing 12 as an argument.

   ```
   def show_value(quantity):
   ```

3. Look at the following function header:

   ```
   def my_function(a, b, c):
   ```

   Now look at the following call to `my_function`:

   ```
   myFunction(3, 2, 1)
   ```

   When this call executes, what value will be assigned to `a`? What value will be assigned to `b`? What value will be assigned to `c`?

4. What will the following program display?

   ```
   def main():
       x = 1
       y = 3.4
       print x, y
       change_us(x, y)
       print x, y

   def change_us(a, b):
       a = 0
       b = 0
       print a, b

   main()
   ```

5. Look at the following function definition:

```
def my_function(a, b, c):
    d = (a + c) / b
    print d
```

   a. Write a statement that calls this function and uses keyword arguments to pass 2 into `a`, 4 into `b`, and 6 into `c`.
   b. What value will be displayed when the function call executes?

Programming Exercises

1. Kilometer Converter

Write a program that asks the user to enter a distance in kilometers, and then converts that distance to miles. The conversion formula is as follows:

$$\text{Miles} = \text{Kilometers} \times 0.6214$$

2. Sale Tax Program Refactoring

Programming Exercise #6 in Chapter 2 was the Sales Tax program. For that exercise you were asked to write a program that calculates and displays the county and state sales tax on a purchase. If you have already written that program, redesign it so the subtasks are in functions. If you have not already written that program, write it using functions.

3. How Much Insurance?

Many financial experts advise that property owners should insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Write a program that asks the user to enter the replacement cost of a building and then displays the minimum amount of insurance he or she should buy for the property.

4. Automobile Costs

Write a program that asks the user to enter the monthly costs for the following expenses incurred from operating his or her automobile: loan payment, insurance, gas, oil, tires, and maintenance. The program should then display the total monthly cost of these expenses, and the total annual cost of these expenses.

5. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. For example, if an acre of land is valued at $10,000, its assessment value is $6,000. The property tax is then 64¢ for each $100 of the assessment value. The tax for the acre assessed at $6,000 will be $38.40. Write a program that asks for the actual value of a piece of property and displays the assessment value and property tax.

6. Body Mass Index

Write a program that calculates and displays a person's body mass index (BMI). The BMI is often used to determine whether a person with a sedentary lifestyle is overweight or underweight for his or her height. A person's BMI is calculated with the following formula:

$$\text{BMI} = \text{weight} \times 703 / \text{height}^2$$

where *weight* is measured in pounds and *height* is measured in inches.

7. Calories from Fat and Carbohydrates

A nutritionist who works for a fitness club helps members by evaluating their diets. As part of her evaluation, she asks members for the number of fat grams and carbohydrate grams that they consumed in a day. Then, she calculates the number of calories that result from the fat, using the following formula:

$$\text{calories from fat} = \text{fat grams} \times 9$$

Next, she calculates the number of calories that result from the carbohydrates, using the following formula:

$$\text{calories from carbs} = \text{carb grams} \times 4$$

The nutritionist asks you to write a program that will make these calculations.

8. Stadium Seating

There are three seating categories at a stadium. For a softball game, Class A seats cost $15, Class B seats cost $12, and Class C seats cost $9. Write a program that asks how many tickets for each class of seats were sold, and then displays the amount of income generated from ticket sales.

9. Paint Job Estimator

A painting company has determined that for every 115 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges $20.00 per hour for labor. Write a program that asks the user to enter the square feet of wall space to be painted and the price of the paint per gallon. The program should display the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

10. Monthly Sales Tax

A retail company must file a monthly sales tax report listing the total sales for the month, and the amount of state and county sales tax collected. The state sales tax rate is 4 percent and the county sales tax rate is 2 percent. Write a program that asks the user to enter the total sales for the month. From this figure, the application should calculate and display the following:

- The amount of county sales tax
- The amount of state sales tax
- The total sales tax (county plus state)

CHAPTER

4 Decision Structures and Boolean Logic

TOPICS

4.1 The `if` Statement
4.2 The `if-else` Statement
4.3 Comparing Strings
4.4 Nested Decision Structures and the `if-elif-else` Statement
4.5 Logical Operators
4.6 Boolean Variables

4.1 The `if` Statement

CONCEPT: The `if` statement is used to create a decision structure, which allows a program to have more than one path of execution. The `if` statement causes one or more statements to execute only when a Boolean expression is true.

A control structure is a logical design that controls the order in which a set of statements execute. So far in this book we have used only the simplest type of control structure: the sequence structure. A sequence structure is a set of statements that execute in the order that they appear. For example, the following code is a sequence structure because the statements execute from top to bottom.

```
name = raw-input('What is your name? ')
age = input('What is your age? ')
print 'Here is the data you entered:'
print 'Name:', name
print 'Age: ', age
```

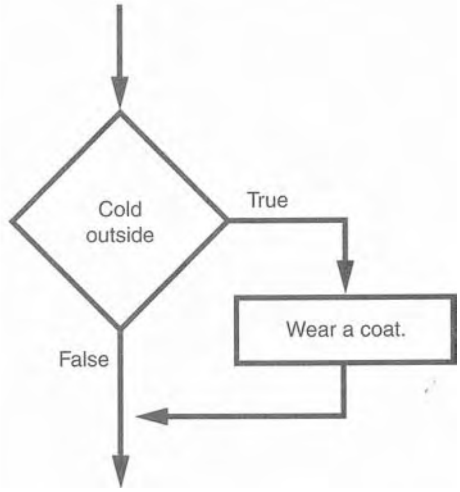
Even in Chapter 3, where you learned about functions, each function contained a block of statements that are executed in the order that they appear. For example, the following function is a sequence structure because the statements in its block execute in the order that they appear, from the beginning of the function to the end.

```
def show_double(value):
    result = value * 2
    print result
```

Although the sequence structure is heavily used in programming, it cannot handle every type of task. This is because some problems simply cannot be solved by performing a set of ordered steps, one after the other. For example, consider a pay calculating program that determines whether an employee has worked overtime. If the employee has worked more than 40 hours, he or she gets paid extra for all the hours over 40. Otherwise, the overtime calculation should be skipped. Programs like this require a different type of control structure: one that can execute a set of statements only under certain circumstances. This can be accomplished with a *decision structure.* (Decision structures are also known as *selection structures.)*

In a decision structure's simplest form, a specific action is performed only if a certain condition exists. If the condition does not exist, the action is not performed. The flowchart shown in Figure 4-1 shows how the logic of an everyday decision can be diagrammed as a decision structure. The diamond symbol represents a true/false condition. If the condition is true, we follow one path, which leads to an action being performed. If the condition is false, we follow another path, which skips the action.

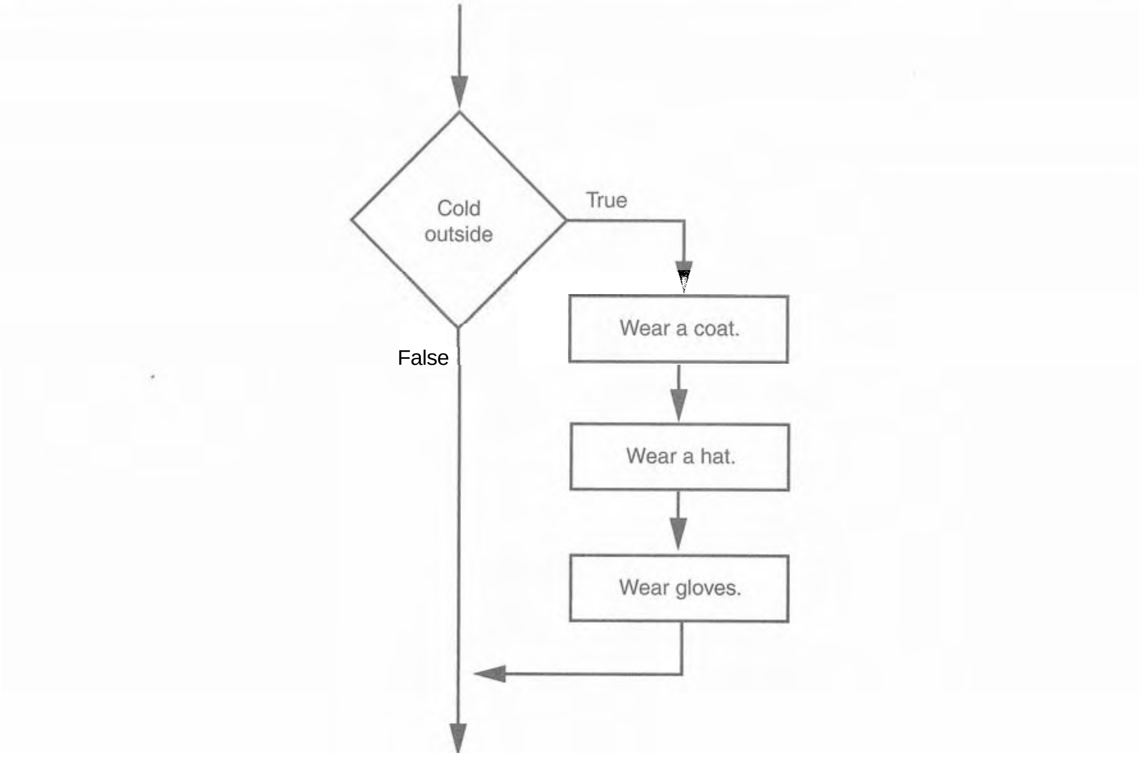
Figure 4-1 A simple decision structure



In the flowchart, the diamond symbol indicates some condition that must be tested. In this case, we are determining whether the condition `Cold outside` is true or false. If this condition is true, the action `wear a coat` is performed. If the condition is false, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition is true.

Programmers call the type of decision structure shown in Figure 4-1 a *single alternative decision structure.* This is because it provides only one alternative path of execution. If the condition in the diamond symbol is true, we take the alternative path. Otherwise, we exit the structure. Figure 4-2 shows a more elaborate example, where three actions are taken only when it is cold outside. It is still a single alternative decision structure, because there is one alternative path of execution.

Figure 4-2 A decision structure that performs three actions if it is cold outside



In Python we use the **`if`** statement to write a single alternative decision structure. Here is the general format of the **`if`** statement:

```
if condition:
    statement
    statement
    etc.
```

For simplicity, we will refer to the first line as the *`if` clause*. The **`if`** clause begins with the word `if`, followed by a *`condition`*, which is an expression that will be evaluated as either true or false. A colon appears after the *`condition`*. Beginning at the next line is a block of statements. (Recall from Chapter **3** that all of the statements in a block must be consistently indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.)

When the **`if`** statement executes, the *`condition`* is tested. If the *`condition`* is true, the statements that appear in the block following the **`if`** clause are executed. If the condition is false, the statements in the block are skipped.

Boolean Expressions and Relational Operators

As previously mentioned, the **`if`** statement tests an expression to determine whether it is true or false. The expressions that are tested by the **`if`** statement are called *Boolean* expressions, named in honor of the English mathematician George Boole. In the 1800s Boole invented a system of mathematics in which the abstract concepts of true and false can be used in computations.

Typically, the Boolean expression that is tested by an `if` statement is formed with a relational operator. A relational operator determines whether a specific relationship exists between two values. For example, the greater than operator (>) determines whether one value is greater than another. The equal to operator (==) determines whether two values are equal. Table 4-1 lists the relational operators that are available in Python.

Table 4-1 Relational operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

The following is an example of an expression that uses the greater than (>) operator to compare two variables, `length` and `width`:

```
length > width
```

This expression determines whether the value referenced by `length` is greater than the value referenced by `width`. If `length` is greater than `width`, the value of the expression is true. Otherwise, the value of the expression is false. The following expression uses the less than operator to determine whether `length` is less than `width`:

```
length < width
```

Table 4-2 shows examples of several Boolean expressions that compare the variables `x` and `y`.

Table 4-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is `x` greater than `y`?
`x < y`	Is `x` less than `y`?
`x >= y`	Is `x` greater than or equal to `y`?
`x <= y`	Is `x` less than or equal to `y`?
`x == y`	Is `x` equal to `y`?
`x != y`	Is `x` not equal to `y`?

The >= and <= Operators

Two of the operators, >= and <=, test for more than one relationship. The >= operator determines whether the operand on its left is greater than or equal to the operand on its right. The <= operator determines whether the operand on its left is less than or equal to the operand on its right.

For example, assume the following:

- **a** is assigned 4
- **b** is assigned **6**
- c is assigned 4

These expressions are true:

```
b >= a
a >= c
a <= c
b <= 10
```

And these expressions are false:

```
a >= 5
b <= a
```

The == Operator

The == operator determines whether the operand on its left is equal to the operand on its right. If the values referenced by both operands are the same, the expression is true. Assuming that **a** is 4, the expression `a == 4` is true and the expression `a == 2` is false.

> **NOTE:** The equality operator is two = symbols together. Don't confuse this operator with the assignment operator, which is one = symbol.

The != Operator

The != operator is the not-equal-to operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the == operator. As before, assuming **a** is 4, **b** is 6, and c is 4, both `a != b` and `b != c` are true because **a** is not equal to b and b is not equal to c. However, `a != c` is false because a is equal to c.

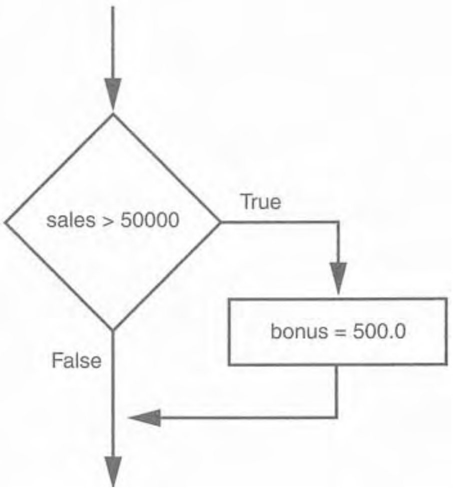
Putting It All Together

Let's look at the following example of the `if` statement:

```
if sales > 50000:
    bonus = 500.0
```

This statement uses the > operator to determine whether **sales** is greater than 50,000. If the expression `sales > 50000` is true, the variable **bonus** is assigned 500.0. If the expression is false, however, the assignment statement is skipped. Figure 4-3 shows a flowchart for this section of code.

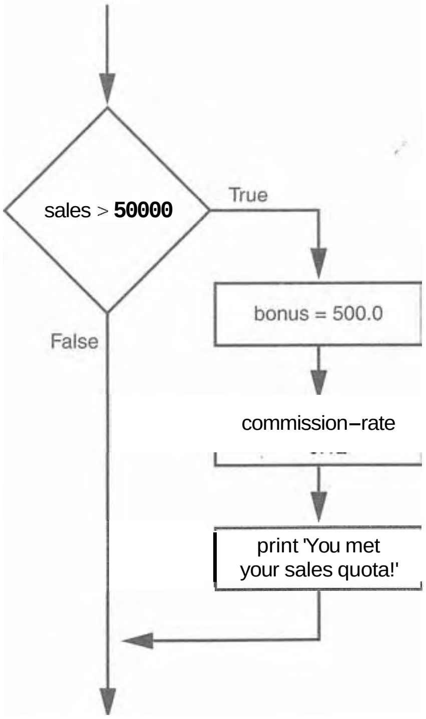
Figure 4-3 Example decision structure


sales > 50000
True
bonus = 500.0
False

The following example conditionally executes three statements. Figure 4-4 shows a flowchart for this section of code.

```
if sales > 50000:
    bonus = 500.0
    commission-rate = 0.12
    print 'You met your sales quota!'
```

Figure 4-4 Example decision structure


sales > 50000
True
bonus = 500.0
False
commission–rate
print 'You met your sales quota!'

The following code uses the == operator to determine whether two values are equal. The expression `balance == 0` will be true if the `balance` variable is assigned 0. Otherwise the expression will be false.

```
if balance == 0:
    # Statements appearing here will
    # be executed only if balance is
    # equal to 0.
```

The following code uses the `!=` operator to determine whether two values are not equal. The expression `choice != 5` will be true if the `choice` variable does not reference the value 5. Otherwise the expression will be false.

```
if choice != 5:
    # Statements appearing here will
    # be executed only if choice is
    # not equal to 5.
```

In the Spotlight:

Using the If-Then Statement

Kathryn teaches a science class and her students are required to take three tests. She wants to write a program that her students can use to calculate their average test score. She also wants the program to congratulate the student enthusiastically if the average is greater than 95. Here is the algorithm in pseudocode:

```
Get the first test score
Get the second test score
Get the third test score
Calculate the average
Display the average
If the average is greater than 95:
    Congratulate the user
```

Program 4-1 shows the code for the program.

Program 4-1 (test_average.py)

```
# This program prompts the user to enter three test
# scores. It displays the average of those scores
# and congratulates the user if the average is 95
# or greater.

def main():
    # Get the three test scores.
    test1 = input('Enter the score for test 1: ')
```

(program continues)

Program 4-1 *(continued)*

```
 9      test2 = input('Enter the score for test 2: ')
10      test3 = input('Enter the score for test 3: ')
11
12      # Calculate the average test score.
13      average = (test1 + test2 + test3) / 3.0
14
15      # Print the average.
16      print 'The average score is', average
17
18      # If the average is 95 or greater,
19      # congratulate the user.
20      if average >= 95:
21          print 'Congratulations!'
22          print 'That is a great average!'
23
24  # Call the main function.
25  main()
```

Program Output (with input shown in bold)

```
Enter the score for test 1: 82 [Enter]
Enter the score for test 2: 76 [Enter]
Enter the score for test 3: 91 [Enter]
The average score is 83.0
```

Program Output (with input shown in bold)

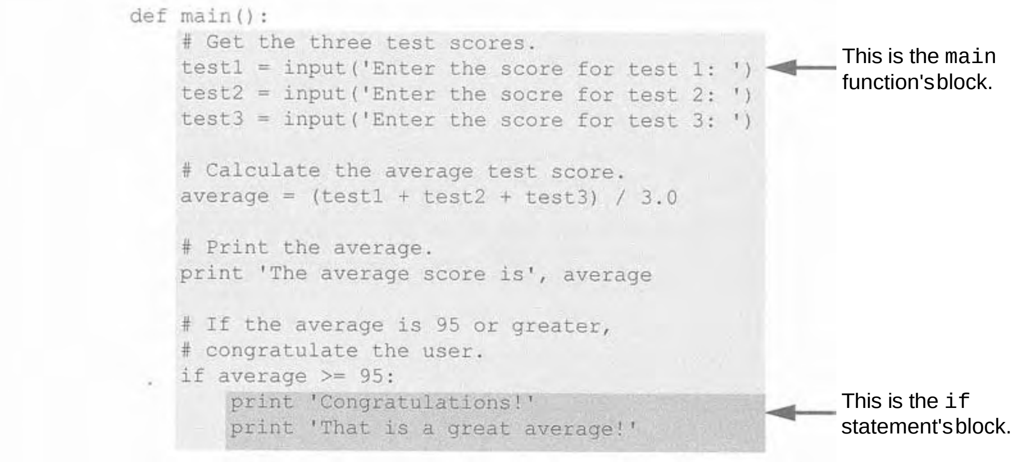
```
Enter the score for test 1: 93 [Enter]
Enter the score for test 2: 99 [Enter]
Enter the score for test 3: 96 [Enter]
The average score is 96.0
Congratulations!
That is a great score.
```

Nested Blocks

Program 4-1 is an example of a program that has a block inside a block. The `main` function has a block (in lines 7 through 22), and inside that block the `if` statement has a block (in lines 21 through 22). This is shown in Figure 4-5.

As you learned in Chapter **3,** Python requires you to indent the statements in a block. When you have a block nested inside a block, the inner block must be further indented. As you can see in Figure 4-5, four spaces are used to indent the `main` function's block, and eight spaces are used to indent the `if` statement's block.

Figure 4-5 Nested blocks



```
def main():
    # Get the three test scores.
    test1 = input('Enter the score for test 1: ')
    test2 = input('Enter the socre for test 2: ')
    test3 = input('Enter the score for test 3: ')

    # Calculate the average test score.
    average = (test1 + test2 + test3) / 3.0

    # Print the average.
    print 'The average score is', average

    # If the average is 95 or greater,
    # congratulate the user.
    if average >= 95:
        print 'Congratulations!'
        print 'That is a great average!'

# Call the main function.
main()
```

This is the `main` function's block.

This is the `if` statement's block.

Checkpoint

4.1 What is a control structure?

4.2 What is a decision structure?

4.3 What is a single alternative decision structure?

4.4 What is a Boolean expression?

4.5 What types of relationships between values can you test with relational operators?

4.6 Write an `if` statement that assigns 0 to x if y is equal to 20.

4.7 Write an `if` statement that assigns 0.2 to `commission` if `sales` is greater than or equal to 10000.

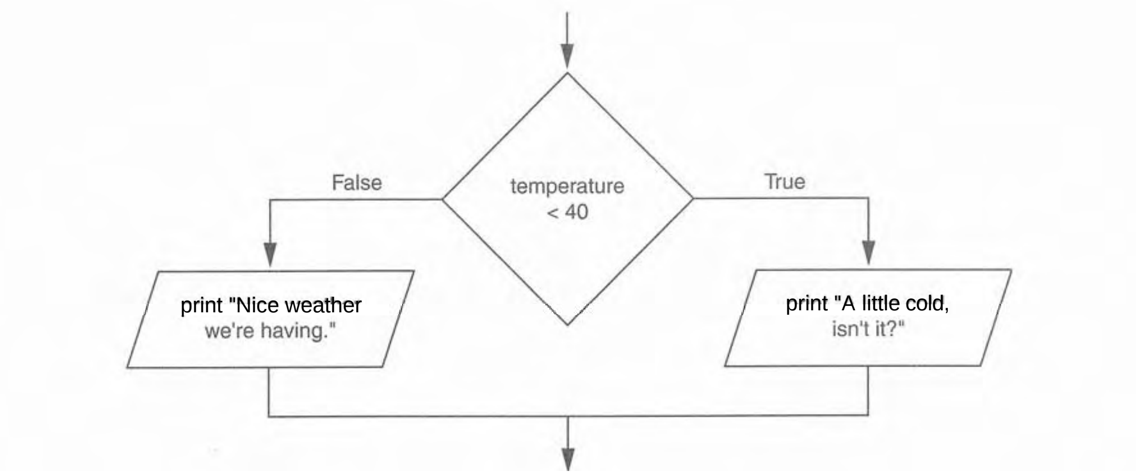
4.2 The `if-else` Statement

CONCEPT: An `if-else` statement will execute one block of statements if its condition is true, or another block if its condition is false.

The previous section introduced the single alternative decision structure (the `if` statement), which has one alternative path of execution. Now we will look at the dual alternative decision structure, which has two possible paths of execution—one path is taken if a condition is true, and the other path is taken if the condition is false. Figure 4-6 shows a flowchart for a dual alternative decision structure.

The decision structure in the flowchart tests the condition `temperature < 40`. If this condition is true, the statement `print "A little cold, isn't it?"` is performed. If the condition is false, the statement `print "Nice weather we're having."` is performed.

Figure 4-6 A dual alternative decision structure



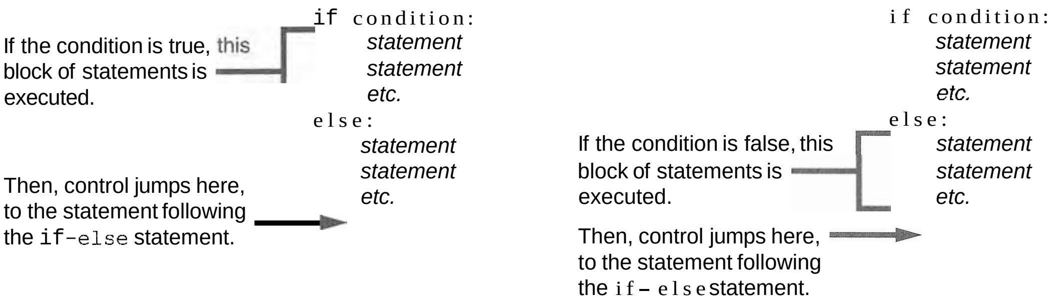
temperature < 40
False
True
print "Nice weather we're having."
print "A little cold, isn't it?"

In code we write a dual alternative decision structure as an `if-else` statement. Here is the general format of the `if-else` statement:

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

When this statement executes, the ***`condition`*** is tested. If it is true, the block of indented statements following the `if` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. If the condition is false, the block of indented statements following the `else` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. This action is described in Figure 4-7.

Figure 4-7 Conditional execution in an `if-else` statement



```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

If the condition is true, this block of statements is executed.

Then, control jumps here, to the statement following the `if-else` statement.

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

If the condition is false, this block of statements is executed.

Then, control jumps here, to the statement following the `if-else` statement.

The following code shows an example of an `if-else` statement. This code matches the flowchart that was shown in Figure 4-5.

```
if temperature < 40:
    print "A little cold, isn't it?"
else:
    print "Nice weather we're having."
```

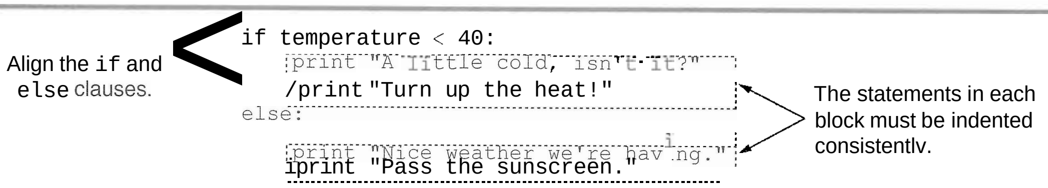
Indentation in the `if-else` Statement

When you write an `if-else` statement, follow these guidelines for indentation:

- Make sure the `if` clause and the `else` clause are aligned.
- The `if` clause and the `else` clause are each followed by a block of statements. Make sure the statements in the blocks are consistently indented.

This is shown in Figure 4-8.

Figure 4-8 Indentation with an `if-else` statement



```
if temperature < 40:
    print "A little cold, isn't it?"
    /print "Turn up the heat!"
else:
    print "Nice weather we're having."
    iprint "Pass the sunscreen."
```

Align the `if` and `else` clauses.

The statements in each block must be indented consistentlv.

In the Spotlight:

Using the `if-else` Statement

Chris owns an auto repair business and has several employees. If any employee works over 40 hours in a week, he pays them 1.5 times their regular hourly pay rate for all hours over 40. He has asked you to design a simple payroll program that calculates an employee's gross pay, including any overtime wages. You design the following algorithm:

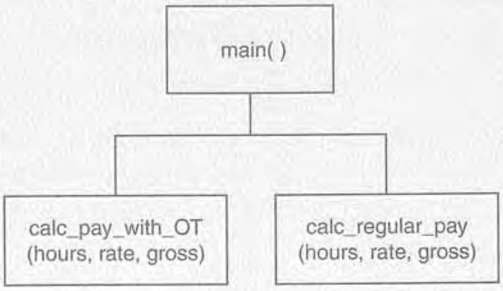
```
Get the number of hours worked.
Get the hourly pay rate.
If the employee worked more than 40 hours:
    Calculate and display the gross pay with overtime.
Else:
    Calculate and display the gross pay as usual.
```

Next, you go through the top-down design process (described in Chapter 3) and create the hierarchy chart shown in Figure 4-9. As shown in the hierarchy chart, there are three functions, summarized as follows:

- `main`—This function will be called when the program starts. It will get the number of hours worked and the hourly pay rate as input from the user. It will then call either the `calc_pay_with_OT` function or the `calc_regular_pay` function to calculate and display the gross pay.

- `calc_pay_with_OT`—This function will calculate and display an employee's pay with overtime.
- `calc_regular_pay`—This function will calculate and display the gross pay for an employee with no overtime.

Figure 4-9 Hierarchy chart


main( )
calc_pay_with_OT (hours, rate, gross)
calc_regular_pay (hours, rate, gross)

The code for the program is shown in Program 4-2. Notice that two global variables, which are used as constants, are created in lines 2 and 3. `BASE_HOURS` is assigned 40, which is the number of hours an employee can work in a week without getting paid overtime. `OT_MULTIPLIER` is assigned 1.5, which is the pay rate multiplier for overtime hours. This means that the employee's hourly pay rate is multiplied by 1.5 for all overtime hours.

Program 4-2 (auto_repair_payroll.py)

```
 1  # Global constants
 2  BASE_HOURS = 40     # Base hours per week
 3  OT_MULTIPLIER = 1.5 # Overtime multiplier
 4
 5  # The main function gets the number of hours worked and
 6  # the hourly pay rate. It calls either the calc_pay_with_OT
 7  # function or the calc_regular_pay function to calculate
 8  # and display the gross pay.
 9  def main():
10      # Get the hours worked and the hourly pay rate.
11      hours_worked = input('Enter the number of hours worked: ')
12      pay_rate = input('Enter the hourly pay rate: ')
13
14      # Calculate and display the gross pay.
15      if hours_worked > BASE_HOURS:
16          calc_pay_with_OT(hours_worked, pay_rate)
17      else:
18          calc_regular_pay(hours_worked, pay_rate)
19
20  # The calc_pay_with_OT function calculates pay with
21  # overtime. It accepts the hours worked and the hourly
```

```
# pay rate as arguments. The gross pay is displayed.
def calc_pay_with_OT(hours, rate):
    # Calculate the number of overtime hours worked.
    overtime_hours = hours - BASE_HOURS

    # Calculate the amount of overtime pay.
    overtime_pay = overtime_hours * rate * OT_MULTIPLIER

    # Calculate the gross pay.
    gross_pay = BASE_HOURS * rate + overtime_pay

    # Display the gross pay.
    print 'The gross pay is $%.2f.' % gross_pay

# The calc_regular_pay function calculates pay with
# no overtime. It accepts the hours worked and the hourly
# pay rate as arguments. The gross pay is displayed.
def calc_regular_pay(hours, rate):
    # Calculate the gross pay.
    gross_pay = hours * rate

    # Display the gross pay.
    print 'The gross pay is $%.2f.' % gross_pay

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the number of hours worked: 40 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $800.00.
```

Program Output (with input shown in bold)

```
Enter the number of hours worked: 50 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $1100.00.
```

Checkpoint use the word filer as manuscript

4.8 How does a dual alternative decision structure work?

4.9 What statement do you use in Python to write a dual alternative decision structure?

4.10 When you write an `if-else` statement, under what circumstances do the statements that appear after the `else` clause execute?

4.3 Comparing Strings

CONCEPT: Python allows you to compare strings. This allows you to create decision structures that test the value of a string.

You saw in the preceding examples how numbers can be compared in a decision structure. You can also compare strings. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 == name2:
    print 'The names are the same.'
else:
    print 'The names are NOT the same.'
```

The == operator compares `name1` and `name2` to determine whether they are equal. Because the strings `'Mary'` and `'Mark'` are not equal, the `else` clause will display the message `'The names are NOT the same.'`

Let's look at another example. Assume the `month` variable references a string. The following code uses the `!=` operator to determine whether the value referenced by `month` is not equal to `'October'`.

```
    if month != 'October':
      print 'This is the wrong time for Octoberfest!'
```

Program 4-3 is a complete program demonstrating how two strings can be compared. The program prompts the user to enter a password and then determines whether the string entered is equal to `'prospero'`.

Program 4-3 (password.py)

```
# This program demonstrates how the == operator can
# be used to compare strings.

def main():
    # Get a password from the user.
    password = raw_input('Enter the password: ')

    # Determine whether the correct password
    # was entered.
    if password == 'prospero':
        print 'Password accepted.'
    else:
        print 'Sorry, that is the wrong password.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the password: ferdinand [Enter]
Sorry, that is the wrong password.
```

Program Output (with input shown in bold)

```
Enter the password: prospero [Enter]
Password accepted.
```

String comparisons are case sensitive. For example, the strings `'saturday'` and `'Saturday'` are not equal because the `"s"` is lowercase in the first string, but uppercase in the second string. The following sample session with Program 4-3 shows what happens when the user enters `Prospero` as the password (with an uppercase `P`).

Program Output (with input shown in bold)

```
Enter the password: Prospero [Enter]
Sorry, that is the wrong password.
```

> TIP: In Chapter 6 you will learn how to manipulate strings so that case-insensitive comparisons can be performed.

Other String Comparisons

In addition to determining whether strings are equal or not equal, you can also determine whether one string is greater than or less than another string. This is a useful capability because programmers commonly need to design programs that sort strings in some order.

Recall from Chapter 1 that computers do not actually store characters, such as A, B, C, and so on, in memory. Instead, they store numeric codes that represent the characters. Chapter 1 mentioned that ASCII (the American Standard Code for Information Interchange) is a commonly used character coding system. You can see the set of ASCII codes in Appendix C, but here are some facts about it:

- The uppercase characters A through Z are represented by the numbers 65 through 90.
- The lowercase characters a through z are represented by the numbers 97 through 122.
- When the digits 0 through 9 are stored in memory as characters, they are represented by the numbers 48 through 57. (For example, the string `'abc123'` would be stored in memory as the codes 97, 98, 99, 49, 50, and 51.)
- A blank space is represented by the number 32.

In addition to establishing a set of numeric codes to represent characters in memory, ASCII also establishes an order for characters. The character "A" comes before the character "B", which comes before the character "C", and so on.

When a program compares characters, it actually compares the codes for the characters. For example, look at the following `if` statement:

```
if 'a' < 'b':
    print 'The letter a is less than the letter b.'
```

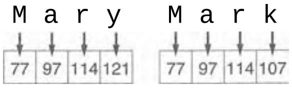
This code determines whether the ASCII code for the character `'a'` is less than the ASCII code for the character `'b'`. The expression `'a' < 'b'` is true because the code for `'a'` is less than the code for `'b'`. So, if this were part of an actual program it would display the message `'The letter a is less than the letter b.'`

Let's look at how strings containing more than one character are typically compared. Suppose a program uses the strings `'Mary'` and `'Mark'` as follows:

```
namel = 'Mary'
name2 = 'Mark'
```

Figure 4-10 shows how the individual characters in the strings `'Mary'` and `'Mark'` would actually be stored in memory, using ASCII codes.

Figure 4-10 Character codes for the strings `'Mary'` and `'Mark'`

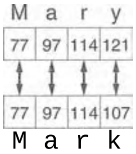


When you use relational operators to compare these strings, the strings are compared character-by-character. For example, look at the following code:

```
namel = 'Mary'
name2 = 'Mark'
if namel > name2:
    print 'Mary is greater than Mark'
else:
    print 'Mary is not greater than Mark'
```

The > operator compares each character in the strings `'Mary'` and `'Mark'`, beginning with the first, or leftmost, characters. This is shown in Figure 4-11.

Figure 4-11 Comparing each character in a string



Here is how the comparison takes place:

1. The `'M'` in `'Mary'` is compared with the `'M'` in `'Mark'`. Since these are the same, the next characters are compared.
2. The `'a'` in `'Mary'` is compared with the `'a'` in `'Mark'`. Since these are the same, the next characters are compared.
3. The `'r'` in `'Mary'` is compared with the `'r'` in `'Mark'`. Since these are the same, the next characters are compared.
4. The `'y'` in `'Mary'` is compared with the `'k'` in `'Mark'`. Since these are not the same, the two strings are not equal. The character `'y'` has a higher ASCII code (121) than `'k'` (107), so it is determined that the string `'Mary'` is greater than the string `'Mark'`.

If one of the strings in a comparison is shorter than the other, only the corresponding characters will be compared. If the corresponding characters are identical, then the shorter string is considered less than the longer string. For example, suppose the strings `'High'` and `'Hi'` were being compared. The string `'Hi'` would be considered less than `'High'` because it is shorter.

Program 4-4 shows a simple demonstration of how two strings can be compared with the < operator. The user is prompted to enter two names and the program displays those two names in alphabetical order.

Program 4-4 (password.py)

```
# This program demonstrates how the < operator can
# be used to compare strings.

def main():
    # Get two names from the user.
    namel = raw-input('Enter a name (last name first): ')
    name2 = raw_input('Enter another name (last name first): ')

    # Display the names in alphabetical order.
    print 'Here are the names, listed alphabetically.'
    if namel < name2:
        print namel
        print name2
    else:
        print name2
        print namel

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a name (last name first): Jones, Richard [Enter]
Enter another name (last name first) Costa, Joan [Enter]
Here are the names, listed alphabetically:
Costa, Joan
Jones, Richard
```

Checkpoint

4.11 What would the following code display?

```
if 'z' < 'a':
    print 'z is less than a.'
else:
    print 'z is not less than a.'
```

4.12 What would the following code display?

```
s1 = 'New York'
s2 = 'Boston'
```

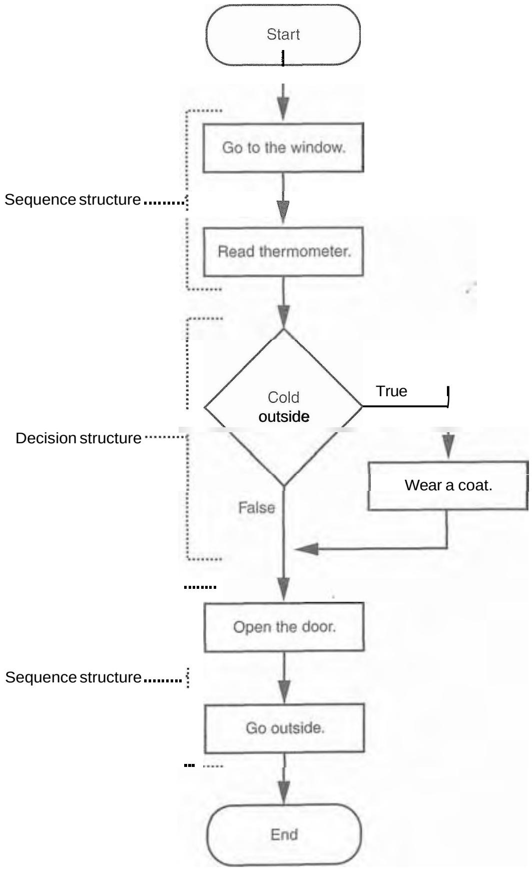
```
if sl > s2:
    print s2
    print sl
else:
    print sl
    print s2
```

4.4 Nested Decision Structures and the `if-elif-else` Statement

CONCEPT: To test more than one condition, a decision structure can be nested inside another decision structure.

In Section 4.1, we mentioned that a control structure determines the order in which a set of statements execute. Programs are usually designed as combinations of different control structures. For example, Figure 4-12 shows a flowchart that combines a decision structure with two sequence structures.

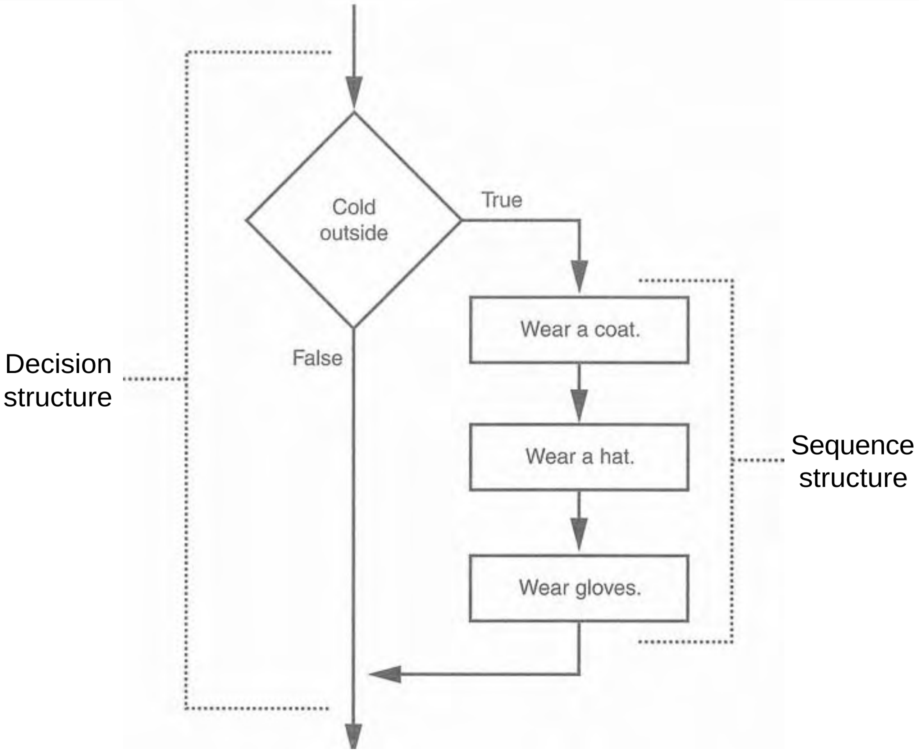
Figure 4-42 Combining sequence structures with a decision structure


Start
Go to the window.
Sequence structure
Read thermometer.
Cold outside
True
Decision structure
Wear a coat.
False
Open the door.
Sequence structure
Go outside.
End

The flowchart in the figure starts with a sequence structure. Assuming you have an outdoor thermometer in your window, the first step is `Go to the window,` and the next step is `Read thermometer.` A decision structure appears next, testing the condition `Cold outside.` If this is true, the action `wear a coat` is performed. Another sequence structure appears next. The step `Open the door` is performed, followed by `Go outside.`

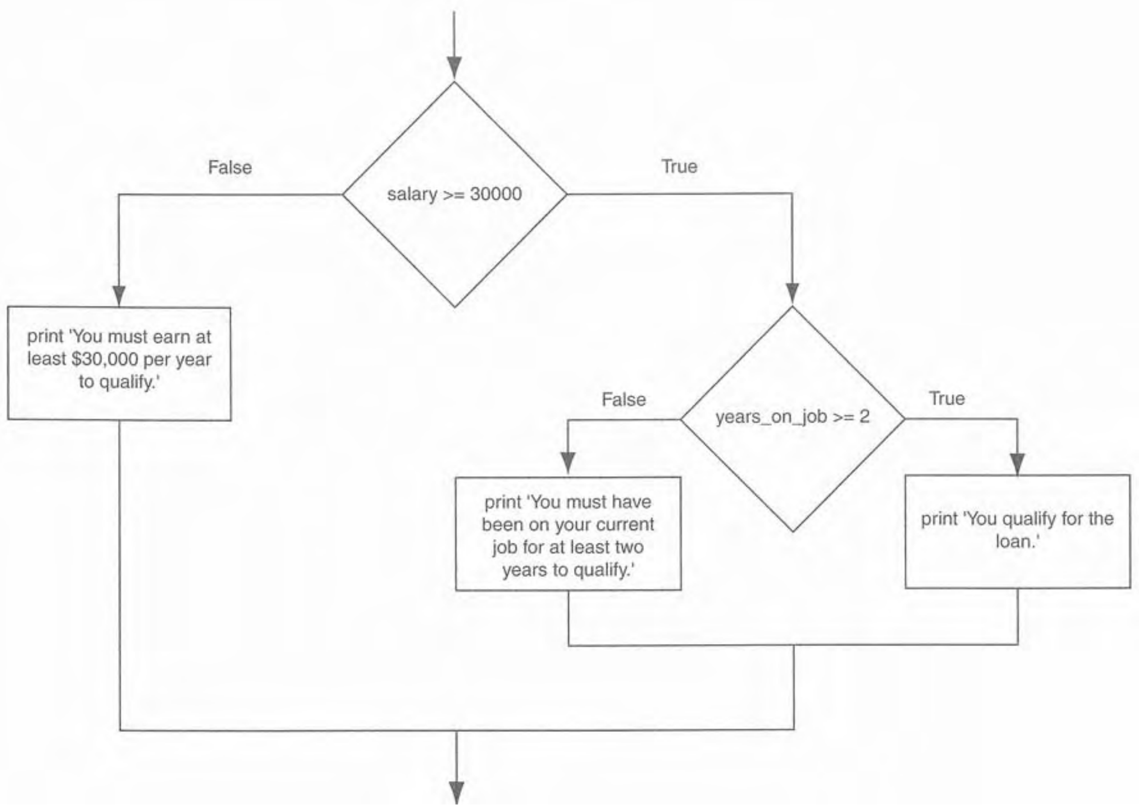
Quite often, structures must be nested inside other structures. For example, look at the partial flowchart in Figure 4-13. It shows a decision structure with a sequence structure nested inside it. The decision structure tests the condition `Cold outside.` If that condition is true, the steps in the sequence structure are executed.

Figure 4-13 A sequence structure nested inside a decision structure



You can also nest decision structures inside other decision structures. In fact, this is a common requirement in programs that need to test more than one condition. For example, consider a program that determines whether a bank customer qualifies for a loan. To qualify, two conditions must exist: (1) the customer must earn at least $30,000 per year, and (2) the customer must have been employed at his or her current job for at least two years. Figure 4-14 shows a flowchart for an algorithm that could be used in such a program. Assume that the `salary` variable is assigned the customer's annual salary, and the `years_on_job` variable is assigned the number of years that the customer has worked on his or her current job.

Figure 4-84 A nested decision structure


salary >= 30000
False
True
print 'You must earn at least $30,000 per year to qualify.'
years_on_job >= 2
False
True
print 'You must have been on your current job for at least two years to qualify.'
print 'You qualify for the loan.'

If we follow the flow of execution, we see that the condition `salary >= 30000` is tested. If this condition is false, there is no need to perform further tests; we know that the customer does not qualify for the loan. If the condition is true, however, we need to test the second condition. This is done with a nested decision structure that tests the condition `years_on_job >= 2`. If this condition is true, then the customer qualifies for the loan. If this condition is false, then the customer does not qualify. Program 4-5 shows the code for the complete program.

Program 4-5 (loan_qualifier.py)

```
# This program determines whether a bank customer
# qualifies for a loan.

def main():
    # Get the customer's annual salary.
    salary = input('Enter your annual salary: ')

    # Get the number of years on the current job.
    years-on-job = input('Enter the number of ' + \
                         'years on your current job: ')
```

```
11
12      # Determine whether the customer qualifies.
13      if salary >= 30000.0:
14          if years-on-job >= 2:
15              print 'You qualify for the loan.'
16          else:
17              print 'You must have been on your current'
18              print 'job for at least two years to qualify.'
19      else:
20          print 'You must earn at least $30,000 per year'
21          print 'to qualify.'
22
23  # Call the main function.
24  main'( )
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 [Enter]
Enter the number of years on your current job: 1 [Enter]
You must have been on your current
job for at least two years to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 [Enter]
Enter the number of years on your current job: 5 [Enter]
You must earn at least $30,000
per year to qualify.
```

Program Output (with input shown in bold)

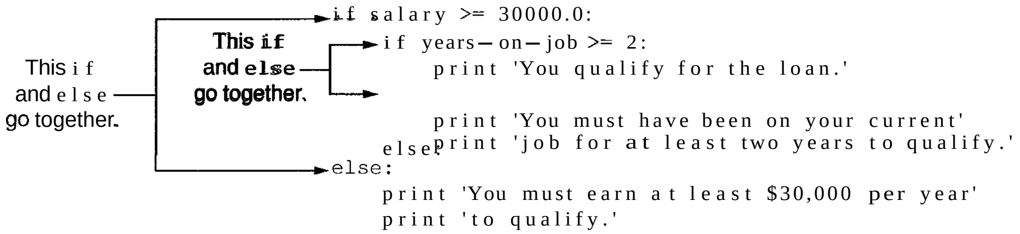
```
Enter your annual salary: 35000 [Enter]
Enter the number of years on your current job: 5 [Enter]
You qualify for the loan.
```

Look at the `if-else` statement that begins in line 13. It tests the condition `salary >= 30000.0`. If this condition is true, the `if-else` statement that begins in line 14 is executed. Otherwise the program jumps to the `else` clause in line 19 and executes the two `print` statements in lines 20 and 21. The program then leaves the decision structure and the `main` function ends.

It's important to use proper indentation in a nested decision structure. Not only is proper indentation required by the Python interpreter, but it also makes it easier for you, the human reader of your code, to see which actions are performed by each part of the structure. Follow these rules when writing nested `if` statements:

- Make sure each `else` clause is aligned with its matching `if` clause. This is shown in Figure 4-15.
- Make sure the statements in each block are consistently indented. The shaded parts of Figure 4-16 show the nested blocks in the decision structure. Notice that each statement in each block is indented the same amount.

Figure 4-15 Alignment of `if` and `else` clauses



```
if salary >= 30000.0:
    if years—on—job >= 2:
        print 'You qualify for the loan.'
    else:
        print 'You must have been on your current'
        print 'job for at least two years to qualify.'
else:
    print 'You must earn at least $30,000 per year'
    print 'to qualify.'
```

Figure 4-16 Nested blocks

```
if salary >= 30000.0:
    If years_on_job >= 7:
        print 'You qualify for the loan.'
    else:
        print 'You must have been on your current'
        print 'job for at least two years to qualify.'
else:
    print 'You must earn at least $30,000 per year'
    print 'to qualify..'
```

Testing a Series of Conditions

In the previous example you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test, and then perform an action depending on which condition is true. One way to accomplish this it to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following In *the Spotlight* section.

In the Spotlight:
Multiple Nested Decision Structures

Dr. Suarez teaches a literature class and uses the following 10 point grading scale for all of his exams:

Test Score	Grade
90 and above	A
80–89	B
70–79	C
60–69	D
Below 60	F

He has asked you to write a program that will allow a student to enter a test score and then display the grade for that score. Here is the algorithm that you will use:

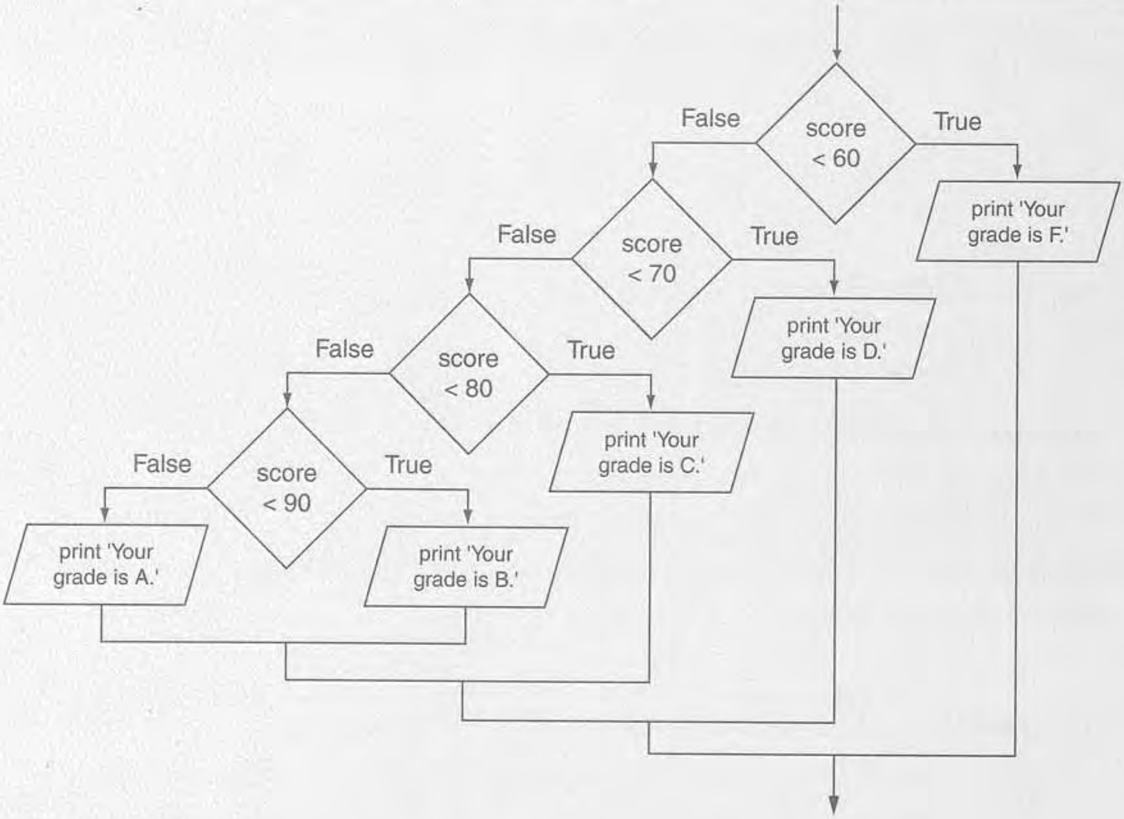
1. Ask the user to enter a test score.

2. Determine the grade in the following manner:

```
If the score is less than 60, then the grade is F.
    Else, if the score is less than 70, then the grade is D.
        Else, if the score is less than 80, then the grade is C.
            Else, if the score is less than 90, then the grade is B.
                Else, the grade is A.
```

You decide that the process of determining the grade will require several nested decision structures, as shown in Figure 4-17. Program 4-6 shows the code for the program. The code for the nested decision structures is in lines 9 through 21.

Figure 4-17 Nested decision structure to determine a grade


False
score < 60
True
print 'Your grade is F.'
False
score < 70
True
print 'Your grade is D.'
False
score < 80
True
print 'Your grade is C.'
False
score < 90
True
print 'Your grade is A.'
print 'Your grade is B.'

Program 4-6 (grader.py)

```
1  # This program gets a numeric test score from the
2  # user and displays the corresponding letter grade.
3
4  def main():
5      # Get a test score from the user.
6      score = input('Enter your test score: ')
```

(program continues)

Program 4-6 *(continued)*

```

    # Determine the grade.
    if score < 60:
        print 'Your grade is F.'
    else:
        if score < 70:
            print 'Your grade is D.'
        else:
            if score < 80:
                print 'Your grade is C.'
            else:
                if score < 90:
                    print 'Your grade is B.'
                else:
                    print 'Your grade is A.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your test score: 78 [Enter]
Your grade is C.
```

Program Output (with input shown in bold)

```
Enter your test score: 84 [Enter]
Your grade is B.
```

The `if-elif-else` Statement

Even though Program 4-6 is a simple example, the logic of the nested decision structure is fairly complex. Python provides a special version of the decision structure known as the `if-elif-else` statement, which makes this type of logic simpler to write. Here is the general format of the `if-elif-else` statement:

```
if condition-1:
    statement
    statement
    etc.
elif condition-2:
    statement
    statement
    etc.
```

```
Insert as many elif clauses as necessary

else:
     statement
     statement
     etc.
```

When the statement executes, `condition_1` is tested. If `condition_1` is true, the block of statements that immediately follow is executed, up to the `elif` clause. The rest of the structure is ignored. If `condition_1` is false, however, the program jumps to the very next `elif` clause and tests `condition_2`. If it is true, the block of statements that immediately follow is executed, up to the next `elif` clause. The rest the structure is then ignored. This process continues until a condition is found to be true, or no more `elif` clauses are left. If no condition is true, the block of statements following the `else` clause is executed.

The following is an example of the `if-elif-else` statement. This code works the same as the nested decision structure in lines 9 through 21 of Program 4-6.

```
if score < 60:
    print 'Your grade is F.'
elif score < 70:
    print 'Your grade is D.'
elif score < 80:
    print 'Your grade is C.'
elif score < 90:
    print 'Your grade is B.'
else:
    print 'Your grade is A.'
```

Notice the alignment and indentation that is used with the `if-elif-else` statement: The `if`, `elif`, and `else` clauses are all aligned, and the conditionally executed blocks are indented.

The `if-elif-else` statement is never required because its logic can be coded with nested `if-else` statements. However, a long series of nested `if-else` statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.
- Because of the required indentation, a long series of nested `if-else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to "wrap around" when printed on paper, making the code even more difficult to read.

The logic of an `if-elif-else` statement is usually easier to follow than a long series of nested `if-else` statements. And, because all of the clauses are aligned in an `if-elif-else` statement, the lengths of the lines in the statement tend to be shorter.

Checkpoint

4.13 Convert the following code to an `if-elif-else` statement:

```
if number == 1:
    print 'One'
```

```
        else:
            if number == 2:
                print 'Two'
            else:
                if number == 3:
                    print 'Three'
                else:
                    print 'Unknown'
```

4.5 Logical Operators

CONCEPT: The logical `and` operator and the logical `or` operator allow you to connect multiple Boolean expressions to create a compound expression. The logical `not` operator reverses the truth of a Boolean expression.

Python provides a set of operators known as logical operators, which you can use to create complex Boolean expressions. Table 4-3 describes these operators.

Table 4-3 Logical operators

Operator	Meaning
`and`	The `and` operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
`or`	The `or` operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
`not`	The `not` operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `not` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

Table 4-4 shows examples of several compound Boolean expressions that use logical operators.

Table 4-4 Compound Boolean expressions using logical operators

Expression	Meaning
`x > y and a < b`	Is x greater than y AND is a less than b?
`x == y or x == z`	Is x equal to y OR is x equal to z?
`not (x > y)`	Is the expression x > y NOT true?

The and Operator

The `and` operator takes two Boolean expressions as operands and creates a compound Boolean expression that is true only when both subexpressions are true. The following is an example of an `if` statement that uses the `and` operator:

```
if temperature < 20 and minutes > 12:
    print 'The temperature is in the danger zone.'
```

In this statement, the two Boolean expressions `temperature < 20` and `minutes > 12` are combined into a compound expression. The `print` statement will be executed only if `temperature` is less than 20 and `minutes` is greater than 12. If either of the Boolean subexpressions is false, the compound expression is false and the message is not displayed.

Table 4-5 shows a truth table for the `and` operator. The truth table lists expressions showing ill the possible combinations of true and false connected with the and operator. The resulting values of the expressions are also shown.

Table 4-5 Truth table for the `and` *operator*

Expression	Value of the Expression
true `and` false	false
false `and` true	false
false `and` false	false
true `and` true	true

As the table shows, both sides of the `and` operator must be true for the operator to return a true value.

The or Operator

The `or` operator takes two Boolean expressions as operands and creates a compound Boolean expression that is true when either of the subexpressions is true. The following is an example of an `if` statement that uses the `or` operator:

```
if,temperature < 20 or temperature > 100:
    print 'The temperature is too extreme'
```

The `print` statement will execute only if `temperature` is less than 20 or `temperature` is greater than 100. If either subexpression is true, the compound expression is true. Table 4-6 shows a truth table for the `or` operator.

Table 4-6 *Truth table for the* `or` operator

Expression	Value of the Expression
true `or` false	true
false `or` true	true
false `or` false	false
true `or` true	true

All it takes for an `or` expression to be true is for one side of the `or` operator to be true. It doesn't matter if the other side is false or true.

Short-Circuit Evaluation

Both the `and` and `or` operators perform *short-circuit evaluation.* Here's how it works with the `and` operator: If the expression on the left side of the `and` operator is false, the expression on the right side will not be checked. Because the compound expression will be false if only one of the subexpressions is false, it would waste CPU time to check the remaining expression. So, when the `and` operator finds that the expression on its left is false, it short-circuits and does not evaluate the expression on its right.

Here's how short-circuit evaluation works with the `or` operator: If the expression on the left side of the `or` operator is true, the expression on the right side will not be checked. Because it is only necessary for one of the expressions to be true, it would waste CPU time to check the remaining expression.

The `not` Operator

The `not` operator is a unary operator that takes a Boolean expression as its operand and reverses its logical value. In other words, if the expression is true, the `not` operator returns false, and if the expression is false, the `not` operator returns true. The following is an `if` statement using the `not` operator:

```
if not(temperature > 100):
    print 'This is below the maximum temperature.'
```

First, the expression `(temperature > 100)` is tested and a value of either true or false is the result. Then the `not` operator is applied to that value. If the expression `(temperature > 100)` is true, the `not` operator returns false. If the expression `(temperature > 100)` is false, the `not` operator returns true. The previous code is equivalent to asking: "Is the temperature not greater than 100?"

NOTE: In this example, we have put parentheses around the expression `temperature > 100`. This is to make it clear that we are applying the `not` operator to the value of the expression `temperature > 100`, not just to the `temperature` variable.

Table 4-7 shows a truth table for the `not` operator.

Table 4-7 Truth table for the `not` operator

Expression	Value of the Expression
`not` true	false
`not` false	true

The Loan Qualifier Program Revisited

In some situations the `and` operator can be used to simplify nested decision structures. For example, recall that the loan qualifier program in Program 4-5 uses the following nested `if-else` statements:

```
if salary >= 30000.0:
    if years-on-job >= 2:
        print 'You qualify for the loan.'
    else:
        print 'You must have been on your current'
        print 'job for at least two years to qualify.'
else:
    print 'You must earn at least $30,000 per year'
    print 'to qualify.'
```

The purpose of this decision structure is to determine that a person's salary is at least $30,000 and that he or she has been at their current job for at least two years. Program 4-7 shows a way to perform a similar task with simpler code.

Program 4-7 (loan_qualifier2.py)

```
# This program determines whether a bank customer
# qualifies for a loan.

def main():
    # Get the customer's annual salary.
    salary = input('Enter your annual salary: ')

    # Get the number of years on the current job.
    years-on-job = input('Enter the number of ' + \
                         'years on your current job: ')

    # Determine whether the customer qualifies.
    if salary >= 30000.0 and years-on-job >= 2:
        print 'You qualify for the loan.'
    else:
        print 'You do not qualify for this loan.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 [Enter]
Enter the number of years on your current job: 1 [Enter]
You do not qualify for this loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 [Enter]
Enter the number of years on your current job: 5 [Enter]
You do not qualify for this loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 [Enter]
Enter the number of years on your current job: 5 [Enter]
You qualify for the loan.
```

The `if-then-else` statement in lines 13 through 16 tests the compound expression `salary >= 30000 and years—on—job >= 2`. If both subexpressions are true, the compound expression is true and the message "You qualify for the loan" is displayed. If either of the subexpressions is false, the compound expression is false and the message "You do not qualify for this loan" is displayed.

NOTE: A careful observer will realize that Program 4-7 is similar to Program 4-5, but it is not equivalent. If the user does not qualify for the loan, Program 4-7 displays only the message "You do not qualify for this loan" whereas Program 4-5 displays one of two possible messages explaining why the user did not qualify.

Yet Another Loan Qualifier Program

Suppose the bank is losing customers to a competing bank that isn't as strict about whom it loans money to. In response, the bank decides to change its loan requirements. Now, customers have to meet only one of the previous conditions, not both. Program 4-8 shows the code for the new loan qualifier program. The compound expression that is tested by the `if-else` statement in line 13 now uses the `or` operator.

Program 4-8 (loan_qualifier3.py)

```
 1  # This program determines whether a bank customer
 2  # qualifies for a loan.
 3
 4  def main():
 5      # Get the customer's annual salary.
 6      salary = input('Enter your annual salary: ')
 7
 8      # Get the number of years on the current job.
 9      years-on-job = input('Enter the number of ' + \
10                           'years on your current job: ')
11
12      # Determine whether the customer qualifies.
13      if salary >= 30000.0 or years-on-job >= 2:
        print 'You qualify for the loan.'
    else:
        print 'You do not qualify for this loan.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 [Enter]
Enter the number of years on your current job: 1 [Enter]
You qualify for the loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 [Enter]
Enter the number of years on your current job: 5 [Enter]
You qualify for the loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary 12000 [Enter]
Enter the number of years on your current job: 1 [Enter]
You do not qualify for this loan.
```

Checking Numeric Ranges with Logical Operators

Sometimes you will need to design an algorithm that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it is best to use the `and` operator. For example, the following `if` statement checks the value in x to determine whether it is in the range of 20 through 40:

```
if x >= 20 and x <= 40:
    print 'The value is in the acceptable range.'
```

The compound Boolean expression being tested by this statement will be true only when x is greater than or equal to 20 and less than or equal to 40. The value in x must be within the range of 20 through 40 for this compound expression to be true.

When determining whether a number is outside a range, it is best to use the `or` operator. The following statement determines whether x is outside the range of 20 through 40:

```
if x < 20 or x > 40:
    print 'The value is outside the acceptable range.'
```

It is important not to get the logic of the logical operators confused when testing for a range of numbers. For example, the compound Boolean expression in the following code would never test true:

```
# This is an error!
if x < 20 and x > 40:
    print 'The value is outside the acceptable range.'
```

Obviously, x cannot be less than 20 and at the same time be greater than 40.

Checkpoint

4.14 What is a compound Boolean expression?

4.15 The following truth table shows various combinations of the values true and false connected by a logical operator. Complete the table by circling T or F to indicate whether the result of such a combination is true or false.

Logical Expression	Result (circle T or F)	
True and False	T	F
True and True	T	F
False and True	T	F
False and False	T	F
True or False	T	F
True or True	T	F
False or True	T	F
False or False	T	F
not True	T	F
not False	T	F

4.16 Assume the variables a = 2, b = 4, and c = 6. Circle the T or F for each of the following conditions to indicate whether its value is true or false.

`a == 4 or b > 2`	T	F
`6 <= c and a > 3`	T	F
`1 != b and c != 3`	T	F
`a >= -1 or a <= b`	T	F
`not (a > 2)`	T	F

4.17 Explain how short-circuit evaluation works with the `and` and `or` operators.

4.18 Write an `if` statement that displays the message "The number is valid" if the value referenced by `speed` is within the range 0 through 200.

4.19 Write an `if` statement that displays the message "The number is not valid" if the value referenced by `speed` is outside the range 0 through 200.

4.6 Boolean Variables

CONCEPT: **A** Boolean variable can reference one of two values: **`True`** or **`False`.** Boolean variables are commonly used as flags, which indicate whether specific conditions exist.

So far in this book we have worked with `int`, `float`, and `str` (string) variables. In addition to these data types, Python also provides a `bool` data type. The `bool` data type allows you to create variables that may reference one of two possible values: `True` or `False`. Here are examples of how we assign values to a `bool` variable:

```
hungry = True
sleepy = False
```

Boolean variables are most commonly used as flags. A flag is a variable that signals when some condition exists in the program. When the flag variable is set to `False`, it indicates the condition does not exist. When the flag variable is set to `True`, it means the condition does exist.

For example, suppose a salesperson has a quota of $50,000. Assuming `sales` references the amount that the salesperson has sold, the following code determines whether the quota has been met:

```
if sales >= 50000.0:
    sales-quota-met = True
else:
    sales-quota-met = False
```

As a result of this code, the `sales_quota—met` variable can be used as a flag to indicate whether the sales quota has been met. Later in the program we might test the flag in the following way:

```
if sales-quota-met:
    print 'You have met your sales quota!'
```

This code displays `'You have met your sales quota!'` if the `bool` variable `sales_quota—met` is `True`. Notice that we did not have to use the == operator to explicitly compare the `sales—quota—met`variable with the value `True`. This code is equivalent to the following:

```
if sales_quota-met == True:
    print 'You have met your sales quota!'
```

Checkpoint

4.20 What values can you assign to a `bool` variable?

4.21 What is a flag variable?

Review Questions

Multiple Choice

1. A __________ structure can execute a set of statements only under certain circumstances.
   a. sequence
   b. circumstantial
   c. decision
   d. Boolean
2. A __________ structure provides one alternative path of execution.
   a. sequence
   b. single alternative decision
   c. one path alternative
   d. single execution decision

3. A(n) ________ expression has a value of either true or false.
   a. binary
   b. decision
   c. unconditional
   d. Boolean
4. The symbols >, <, and == are all ________ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary
5. A(n) ________ structure tests a condition and then takes one path if the condition is true, or another path if the condition is false.
   a. `if` statement
   b. single alternative decision
   c. dual alternative decision
   d. sequence
6. You use a(n) ________ statement to write a single alternative decision structure.
   a. `test-jump`
   b. `if`
   c. `if-else`
   d. `if-call`
7. You use a(n) ________ statement to write a dual alternative decision structure.
   a. `test-jump`
   b. `if`
   c. `if-else`
   d. `if-call`
8. `and`, `or`, and `not` are ________ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary
9. A compound Boolean expression created with the ________ operator is true only if both of its subexpressions are true.
   a. `and`
   b. `or`
   c. `not`
   d. `both`
10. A compound Boolean expression created with the ________ operator is true if either of its subexpressions is true.
    a. `and`
    b. `or`
    c. `not`
    d. `either`

11. The __________ operator takes a Boolean expression as its operand and reverses its logical value.
    a. `and`
    b. `or`
    c. `not`
    d. `either`
12. A __________ is a Boolean variable that signals when some condition exists in the program.
    a. flag
    b. signal
    c. sentinel
    d. siren

True or False

1. You can write any program using only sequence structures.
2. A program can be made of only one type of control structure. You cannot combine structures.
3. A single alternative decision structure tests a condition and then takes one path if the condition is true, or another path if the condition is false.
4. A decision structure can be nested inside another decision structure.
5. A compound Boolean expression created with the `and` operator is true only when both subexpressions are true.

Short Answer

1. Explain what is meant by the term "conditionally executed."
2. You need to test a condition and then execute one set of statements if the condition is true. If the condition is false, you need to execute a different set of statements. What structure will you use?
3. Briefly describe how the `and` operator works.
4. Briefly describe how the `or` operator works.
5. When determining whether a number is inside a range, which logical operator is it best to use?
6. What is a flag and how does it work?

Algorithm Workbench

1. Write an `if` statement that assigns 20 to the variable `y` and assigns 40 to the variable `z` if the variable `x` is greater than 100.
2. Write an `if` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10.
3. Write an `if-else` statement that assigns 0 to the variable `b` if the variable `a` is less than 10. Otherwise, it should assign 99 to the variable `b`.
4. The following code contains several nested `if-else` statements. Unfortunately, it was written without proper alignment and indentation. Rewrite the code and use the proper conventions of alignment and indentation.

```
if score < 60:
print 'Your grade is F.'
```

```
else:
if score < 70:
print 'Your grade is D.'
else:
if score < 80:
print 'Your grade is C.'
else:
if score < 90:
print 'Your grade is B.'
else:
print 'Your grade is A.'
```

5. Write nested decision structures that perform the following: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of `amount1` and `amount2`.
6. Write an `if-else` statement that displays `'Speed is normal'` if the `speed variable` is within the range of 24 to 56. If the `speed` variable's value is outside this range, display `'Speed is abnormal'`.
7. Write an `if-else` statement that determines whether the `points` variable is outside the range of 9 to 51. If the variable's value is outside this range it should display "Invalid points." Otherwise, it should display "Valid points."

Programming Exercises

1. Roman Numerals

Write a program that prompts the user to enter a number within the range of 1 through 10. The program should display the Roman numeral version of that number. If the number is outside the range of 1 through 10, the program should display an error message. The following table shows the Roman numerals for the numbers 1 through 10:

Number	Roman Numeral
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

2. Areas of Rectangles

The area of a rectangle is the rectangle's length times its width. Write a program that asks for the length and width of two rectangles. The program should tell the user which rectangle has the greater area, or if the areas are the same.

3. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in newtons. If you know the amount of mass of an object in kilograms, you can calculate its weight in newtons with the following formula:

$$weight = mass \times 9.8$$

Write a program that asks the user to enter an object's mass, and then calculates its weight. If the object weighs more than 1000 newtons, display a message indicating that it is too heavy. If the object weighs less than 10 newtons, display a message indicating that it is too light.

4. Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month and displays the number of points awarded.

5. Software Sales

A software company sells a package that retails for $99. Quantity discounts are given according to the following table:

Quantity	Discount
10–19	20%
20–49	30%
50–99	40%
100 or more	50%

Write a program that asks the user to enter the number of packages purchased. The program should then display the amount of the discount (if any) and the total amount of the purchase after the discount.

6. Shipping Charges

The Fast Freight Shipping Company charges the following rates:

Weight of Package	Rate per Pound
2 pounds or less	$1.10
Over 2 pounds but not more than 6 pounds	$2.20
Over 6 pounds but not more than 10 pounds	$3.70
Over 10 pounds	$3.80

Write a program that asks the user to enter the weight of a package and then displays the shipping charges.

7. Body Mass Index Program Enhancement

In programming Exercise #6 in Chapter 3 you were asked to write a program that calculates a person's body mass index (BMI). Recall from that exercise that the BMI is often used to determine whether a person with a sedentary lifestyle is overweight or underweight for their height. A person's BMI is calculated with the formula

$$BMI = weight \times 703 \, / \, height^2$$

where *weight* is measured in pounds and *height* is measured in inches. Enhance the program so it displays a message indicating whether the person has optimal weight, is underweight, or is overweight. A sedentary person's weight is considered to be optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered to be underweight. If the BMI value is greater than 25, the person is considered to be overweight.

8. Time Calculator

Write a program that asks the user to enter a number of seconds, and works as follows:

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.

CHAPTER

5 Repetition Structures

TOPICS

- 5.1 Introduction to Repetition Structures
- 5.2 The `while` Loop: a Condition-Controlled Loop
- 5.3 The `for` Loop: a Count-Controlled Loop
- 5.4 Calculating a Running Total
- 5.5 Sentinels
- 5.6 Input Validation Loops
- 5.7 Nested Loops

5.1 Introduction to Repetition Structures

CONCEPT: A repetition structure causes a statement or set of statements to execute repeatedly.

Programmers commonly have to write code that performs the same task over and over. For example, suppose you have been asked to write a program that calculates a 10 percent sales commission for several sales people. Although it would not be a good design, one approach would be to write the code to calculate one sales person's commission, and then repeat that code for each sales person. For example, look at the following:

```
# Get a salesperson's sales and commission rate.
sales = input('Enter the amount of sales: ')
comm_rate = input('Enter the commission rate: ')

# Calculate the commission.
commission = sales * comm_rate

# Display the commission.
print 'The commission is $%.2f.' % commission

# Get another salesperson's sales and commission rate.
sales = input('Enter the amount of sales: ')
comm_rate = input('Enter the commission rate: ')

# Calculate the commission.
commission = sales * comm_rate
```

```
# Display the commission.
print 'The commission is $%.2f.' % commission

# Get another salesperson's sales and commission rate.
sales = input('Enter the amount of sales: ')
comm_rate = input('Enter the commission rate: ')

# Calculate the commission.
commission = sales * comm_rate

# Display the commission.
print 'The commission is $%.2f.' % commission
```

And this code goes on and on . . .

As you can see, this code is one long sequence structure containing a lot of duplicated code. There are several disadvantages to this approach, including the following:

- The duplicated code makes the program large.
- Writing a long sequence of statements can be time consuming.
- If part of the duplicated code has to be corrected or changed then the correction or change has to be done many times.

Instead of writing the same sequence of statements over and over, a better way to repeatedly perform an operation is to write the code for the operation once, and then place that code in a structure that makes the computer repeat it as many times as necessary. This can be done with a *repetition structure,* which is more commonly known as a *loop*.

Condition-Controlled and Count-Controlled Loops

In this chapter, we will look at two broad categories of loops: condition-controlled and count-controlled. A *condition-controlled loop* uses a true/false condition to control the number of times that it repeats. A *count-controlled loop* repeats a specific number of times. In Python you use the `while` statement to write a condition-controlled loop, and you use the `for` statement to write a count-controlled loop. In this chapter, we will demonstrate how to write both types of loops.

Checkpoint

5.1 What is a repetition structure?

5.2 What is a condition-controlled loop?

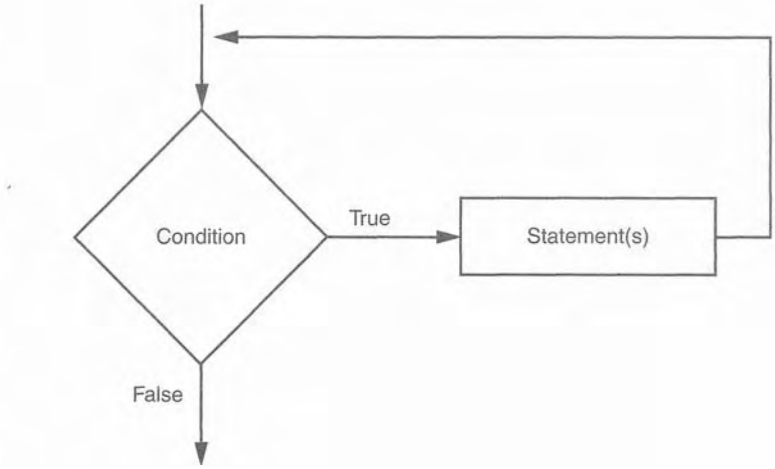
5.3 What is a count-controlled loop?

5.2 The `while` Loop: a Condition-Controlled Loop

CONCEPT: A condition-controlled loop causes a statement or set of statements to repeat as long as a condition is true. In Python you use the `while` statement to write a condition-controlled loop.

The `while` loop gets its name from the way it works: while a condition is true, do some task. The loop has two parts: (1) a condition that is tested for a true or false value, and (2) a statement or set of statements that is repeated as long as the condition is true. Figure 5-1 shows the logic of a `while` loop.

Figure 5-1 The logic of a `while` loop



The diamond symbol represents the condition that is tested. Notice what happens if the condition is true: one or more statements are executed and the program's execution flows back to the point just above the diamond symbol. The condition is tested again, and if it is true, the process repeats. If the condition is false, the program exits the loop. In a flowchart, you will always recognize a loop when you see a flow line going back to a previous part of the flowchart.

Here is the general format of the `while` loop in Python:

```
while condition:
    statement
    statement
    etc.
```

For simplicity, we will refer to the first line as the *`while`* clause. The `while` clause begins with the word `while`, followed by a Boolean *`condition`* that will be evaluated as either true or false. A colon appears after the *`condition`*. Beginning at the next line is a block of statements. (Recall from Chapter 3 that all of the statements in a block must be consistently indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.)

When the `while` loop executes, the *`condition`* is tested. If the *`condition`* is true, the statements that appear in the block following the `while` clause are executed, and then the loop starts over. If the *`condition`* is false, the program exits the loop. Program 5-1 shows how we might use a `while` loop to write the commission calculating program that was described at the beginning of this chapter.

Program 5-1 (commission.py)

```
 1  # This program calculates sales commissions.
 2  def main():
 3      # Create a variable to control the loop.
 4      keep-going = 'y'
 5
 6      # Calculate a series of commissions.
 7      while keep-going == 'y':
 8          # Get a salesperson's sales and commission rate.
 9          sales = input('Enter the amount of sales: ')
10          comm_rate = input('Enter the commission rate: ')
13
12          # Calculate the commission.
13          commission = sales * comm_rate
14
15          # Display the commission.
16          print 'The commission is $%.2f.' % commission
17
18          # See if the user wants to do another one.
19          keep-going = raw-input('Do you want to calculate another ' + \
20                                 'commission (Enter y for yes): ')
21
22  # Call the main function.
23  main()
```

Program Output (with input shown in bold)

```
Enter the amount of sales: 10000.00 [Enter]
Enter the commission rate: 0.10 [Enter]
The commission is $1000.00.
Do you want to calculate another commission (Enter y for yes): y [Enter]
Enter the amount of sales: 20000.00 [Enter]
Enter the commission rate: 0.15 [Enter]
The commission is $3000.00.
Do you want to calculate another commission (Enter y for yes): y [Enter]
Enter the amount of sales: 12000.00 [Enter]
Enter the commission rate: 0.10 [Enter]
The commission is $1200.00.
Do you want to calculate another commission (Enter y for yes): n [Enter]
```

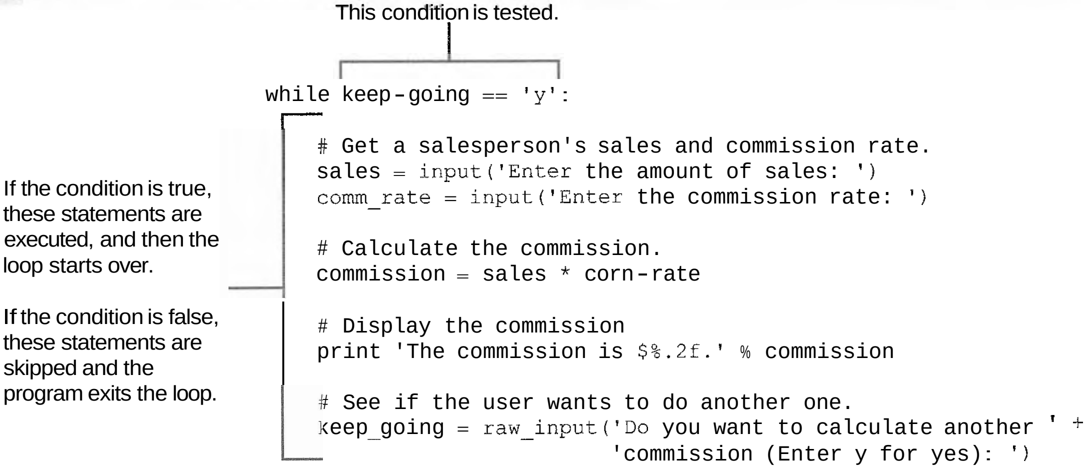
In line 4 we use an assignment statement to create a variable named `keep_going`. Notice that the variable is assigned the value `'y'`. This initialization value is important, and in a moment you will see why.

Line 7 is the beginning of a `while` loop, which starts like this:

```
while keep_going == 'y':
```

Notice the condition that is being tested: `keep_going == 'y'`. The loop tests this condition, and if it is true, the statements in lines 8 through 20 are executed. Then, the loop starts over at line 7. It tests the expression `keep-going == 'y'` and if it is true, the statements in lines 8 through 20 are executed again. This cycle repeats until the expression `keepgoing == 'y'` is tested in line 7 and found to be false. When that happens, the program exits the loop. This is illustrated in Figure 5-2.

Figure 5-2 The `while` loop



This condition is tested.

If the condition is true, these statements are executed, and then the loop starts over.

If the condition is false, these statements are skipped and the program exits the loop.

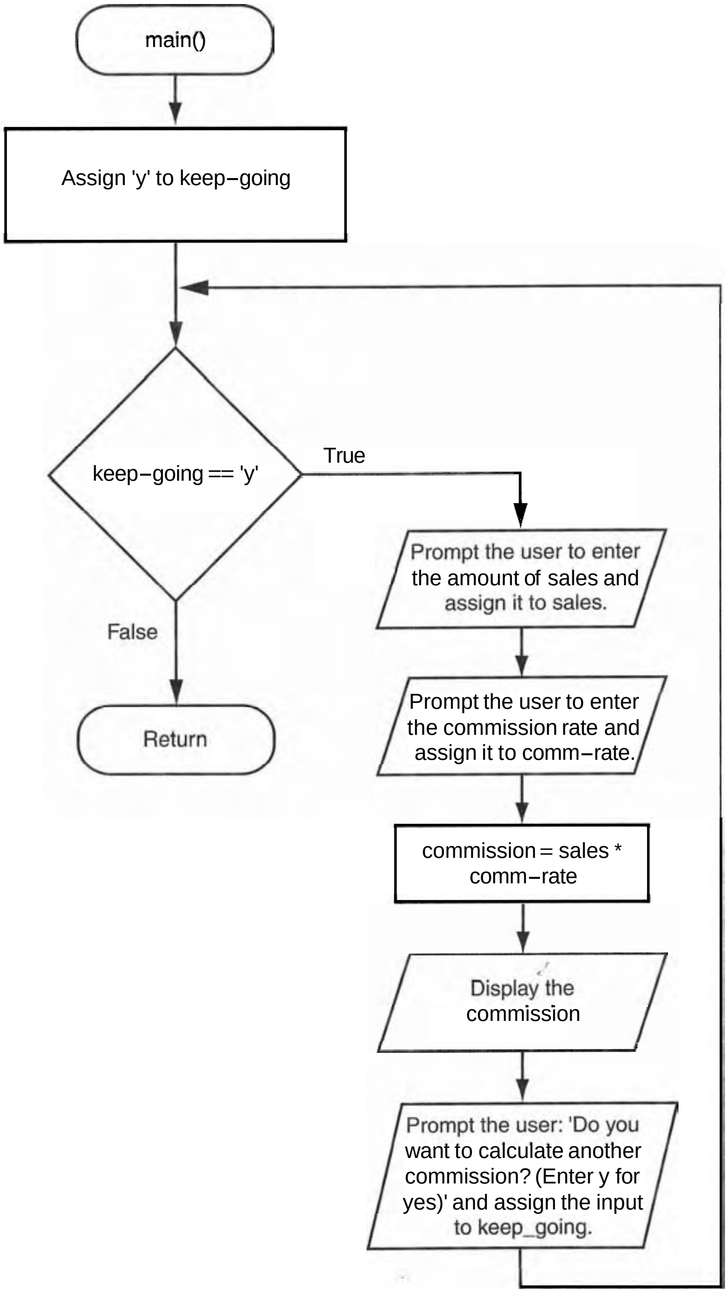
```
while keep-going == 'y':

    # Get a salesperson's sales and commission rate.
    sales = input('Enter the amount of sales: ')
    comm_rate = input('Enter the commission rate: ')

    # Calculate the commission.
    commission = sales * corn-rate

    # Display the commission
    print 'The commission is $%.2f.' % commission

    # See if the user wants to do another one.
    keep_going = raw_input('Do you want to calculate another ' +
                           'commission (Enter y for yes): ')
```

In order for this loop to stop executing, something has to happen inside the loop to make the expression `keep-going == 'y'` false. The statement in lines 19 through 20 take care of this. This statement displays the prompt "Do you want to calculate another commission (Enter y for yes)." The value that is read from the keyboard is assigned to the `keep-going` variable. If the user enters y (and it must be a lowercase y), then the expression `keep-going == 'y'` will be true when the loop starts over. This will cause the statements in the body of the loop to execute again. But if the user enters anything other than lowercase y, the expression will be false when the loop starts over, and the program will exit the loop.

Now that you have examined the code, look at the program output in the sample run. First, the user entered 10000.00 for the sales and 0.10 for the commission rate. Then, the program displayed the commission for that amount, which is $1000.00. Next the user is prompted "Do you want to calculate another commission? (Enter y for yes)." The user entered y, and the loop started the steps over. In the sample run, the user went through this process three times. Each execution of the body of a loop is known as an iteration. In the sample run, the loop iterated three times.

Figure 5-3 shows a flowchart for the `main` function. In the flowchart we have a repetition structure, which is the `while` loop. The condition `keep-going == 'y'` is tested, and if it is true a series of statements are executed and the flow of execution returns to the point just above the conditional test.

Figure 5-3 Flowchart for Program 5-1


main()
Assign 'y' to keep–going
keep–going == 'y'
True
False
Return
Prompt the user to enter the amount of sales and assign it to sales.
Prompt the user to enter the commission rate and assign it to comm–rate.
commission = sales * comm–rate
Display the commission
Prompt the user: 'Do you want to calculate another commission? (Enter y for yes)' and assign the input to keep_going.

The `while` Loop is a Pretest Loop

The `while` loop is known as a ***pretest*** loop, which means it tests its condition *before* performing an iteration. Because the test is done at the beginning of the loop, you usually have to perform some steps prior to the loop to make sure that the loop executes at least once. For example, the loop in Program 5-1 starts like this:

```
while keep-going == 'y':
```

The loop will perform an iteration only if the expression `keep_going == 'y'` is true. This means that (a) the `keep_going` variable has to exist, and (b) it has to reference the value `'y'`. To make sure the expression is true the first time that the loop executes, we assigned the value `'y'` to the `keep_going` variable in line 4 as follows:

```
keep-going = 'y'
```

By performing this step we know that the condition `keep_going == 'y'` will be true the first time the loop executes. This is an important characteristic of the `while` loop: it will never execute if its condition is false to start with. In some programs, this is exactly what you want. The following In the *Spotlight* section gives an example.

In the Spotlight:

Designing a Program with a `while` Loop

A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. Get the substance's temperature.
2. Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:
    a. Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.
    b. Get the substance's temperature.
3. After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.

After reviewing this algorithm, you realize that steps 2(a) and 2(b) should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is false. Program 5-2 shows the code for the program.

Program 5-2 (temperature.py)

```
# This program assists a technician in the process
# of checking a substance's temperature.

# MAX_TEMP is used as a global constant for
# the maximum temperature.
MAX_TEMP = 102.5

# The main function
def main():
    # Get the substance's temperature.
    temperature = input("Enter the substance's Celsius temperature: ")

    # As long as necessary, instruct the user to
    # adjust the thermostat.
    while temperature > MAX_TEMP:
        print 'The temperature is too high.'
        print 'Turn the thermostat down and wait'
        print '5 minutes. Then take the temperature'
        print 'again and enter it.'
        temperature = input('Enter the new Celsius temperature: ')

    # Remind the user to check the temperature again
    # in 15 minutes.
    print 'The temperature is acceptable.'
    print 'Check it again in 15 minutes.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the substance's Celsius temperature: 104.7 [Enter]
The temperature is too high.
Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.
Enter the new Celsius temperature: 103.2 [Enter]
The temperature is too high.
Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.
Enter the new Celsius temperature: 102.1 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

Program Output (with input shown in bold)

```
Enter the substance's Celsius temperature: 102.1 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test condition false. The loop in Program 5-1 stops when the expression `keepgoing == 'y'` is false. If a loop does not have a way of stopping, it is called an infinite loop. An ***infinite*** loop continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false. In most circumstances you should avoid writing infinite loops.

Program 5-3 demonstrates an infinite loop. This is a modified version of the commission calculating program shown in Program 5-1. In this version, we have removed the code that modifies the `keep_going` variable in the body of the loop. Each time the expression `keep—going == 'y'` is tested in line 7, `keep_going` will reference the string 'y'. As a consequence, the loop has no way of stopping.

Program 5-3 (infinite.py)

```
 1  # This program demonstrates an infinite loop.
 2  def main():
 3      # Create a variable to control the loop.
 4      keep-going = 'y'
 5
 6      # Warning! Infinite loop!
 7      while keep-going == 'y':
 8          # Get a salesperson's sales and commission rate.
 9          sales = input('Enter the amount of sales: ')
10          corm-rate = input('Enter the commission rate: ')
11
12          # Calculate the commission.
13          commission = sales * comm_rate
14
15          # Display the commission.
16          print 'The commission is $%.2f.' % commission
17
18  # Call the main function.
19  main()
```

Calling Functions in a Loop

Functions can be called from statements in the body of a loop. In fact, such code in a loop often improves the design. For example, in Program 5-1, the statements that get the amount of sales, calculate the commission, and display the commission can easily be placed in a function. That function can then be called in the loop. Program 5-4 shows how this might be done.

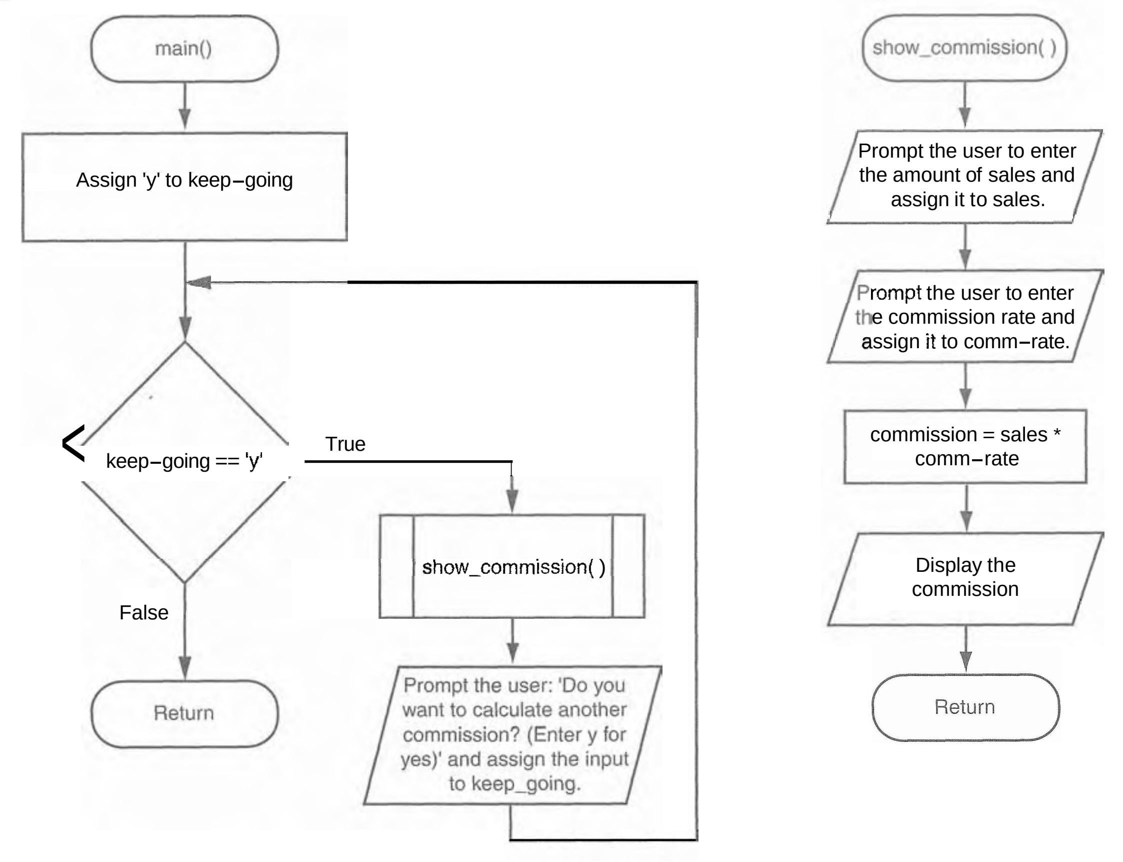
This program has a `main` function, which is called when the program runs, and a `show_commission` function that handles all of the steps related to calculating and displaying a commission. Figure 5-4 shows flowcharts for the `main` and `show_commission` functions.

Program 5-4 (commission2.py)

```
# This program calculates sales commissions.
def main():
    # Create a variable to control the loop.
    keep-going = 'y'

    # Calculate a series of commissions.
    while keep-going == 'y':
        # Call the show~commissionfunction to
        # display a salesperson's commission.
        show_commission()

        # See if the user wants to do another one.
        keep-going = raw-input('Do you want to calculate another ' + \
                                'commission (Enter y for yes): ')

# The show~commissionfunction gets the amount of
# sales and the commission rate, and then displays
# the amount of commission.
def show_commission():
    # Get a salesperson's sales and commission rate.
    sales = input('Enter the amount of sales: ')
    comm_rate = input('Enter the commission rate: ')

    # Calculate the commission.
    commission = sales * comm_rate

    # Display the commission.
    print 'The commission is $%.2f.' % commission

# Call the main function.
main()
```

The output of this program is the same as that of Program 5-1

Figure 5-4 Flowcharts for the `main` and `show_commission` functions


main()
Assign 'y' to keep-going
keep-going == 'y'
True
False
show_commission( )
Prompt the user: 'Do you want to calculate another commission? (Enter y for yes)' and assign the input to keep_going.
Return
show_commission( )
Prompt the user to enter the amount of sales and assign it to sales.
Prompt the user to enter the commission rate and assign it to comm-rate.
commission = sales * comm-rate
Display the commission
Return

Checkpoint

5.4 What is a loop iteration?

5.5 Does the `while` loop test its condition before or after it performs an iteration?

5.6 How many times will `'Hello World'` be printed in the following program?

```
count = 10
while count < 1:
    print 'Hello World'
```

5.7 What is an infinite loop?

5.3 The for Loop: a Count-Controlled Loop

CONCEPT: A count-controlled loop iterates a specific number of times. In Python you use the `for` statement to write a count-controlled loop.

As mentioned at the beginning of this chapter, a count-controlled loop iterates a specific number of times. Count-controlled loops are commonly used in programs. For example, suppose a business is open six days per week, and you are going to write a program that calculates the total sales for a week. You will need a loop that iterates exactly six times. Each time the loop iterates, it will prompt the user to enter the sales for one day.

You use the `for` statement to write a count-controlled loop. In Python, the `for` statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence. Here is the general format:

```
for variable in [value1, value2, etc.]:
     statement
     statement
     etc.
```

We will refer to the first line as the *`for` clause*. In the `for` clause, *variable* is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value. (In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a ***list***. In Chapter 8 you will learn more about lists.) Beginning at the next line is a block of statements that is executed each time the loop iterates.

The `for` statement executes in the following manner: The *variable* is assigned the first value in the list, and then the statements that appear in the block are executed. Then, *variable* is assigned the next value in the list, and the statements in the block are executed again. This continues until *variable* has been assigned the last value in the list. Program 5-5 shows a simple example that uses a `for` loop to display the numbers 1 through 5.

Program 5-5 (simple-loop1.py)

```
 1  # This program demonstrates a simple for loop
 2  # that uses a list of numbers.
 3
 4  def main():
 5      print 'I will display the numbers 1 through 5.'
 6      for num in [1, 2, 3, 4, 5]:
 7          print num
 8
 9  # Call the main function.
10  main()
```

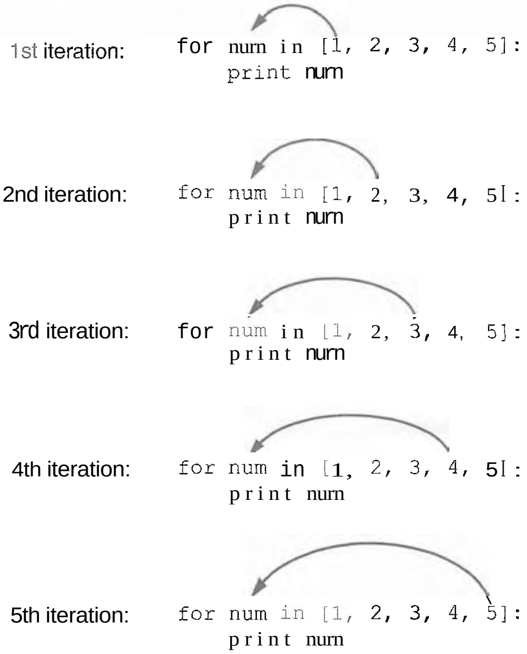
Program Output

```
I will display the numbers 1 through 5.
1
2
3
4
5
```

The first time the `for` loop iterates, the `num` variable is assigned the value 1 and then the `print` statement in line 7 executes (displaying the value 1). The next time the loop iterates, num is assigned the value 2, and the `print` statement executes (displaying the value 2). This process continues, as shown in Figure 5-5, until num has been assigned the last value in the list. Because the list contains five values, the loop will iterate five times.

Python programmers commonly refer to the variable that is used in the `for` clause as the *target variable* because it is the target of an assignment at the beginning of each loop iteration.

Figure 5-5 The `for` loop


1st iteration:
for num in [1, 2, 3, 4, 5]:
print num
2nd iteration:
for num in [1, 2, 3, 4, 5]:
print num
3rd iteration:
for num in [1, 2, 3, 4, 5]:
print num
4th iteration:
for num in [1, 2, 3, 4, 5]:
print num
5th iteration:
for num in [1, 2, 3, 4, 5]:
print num

The values that appear in the list do not have to be a consecutively ordered series of numbers. For example, Program 5-6 uses a `for` loop to display a list of odd numbers. There are five numbers in the list, so the loop iterates five times.

Program 5-6 (simple_loop2.py)

```
# This program also demonstrates a simple for
# loop that uses a list of numbers.

def main():
    print 'I will display the odd numbers 1 through 9.'
    for num in [1, 3, 5, 7, 9]:
        print num

# Call the main function.
main()
```

Program Output

```
I will display the odd numbers 1 through 9.
1
3
5
7
9
```

Program 5-7 shows another example. In this program the `for` loop iterates over a list of strings. Notice that the list (in line 5) contains the three strings 'Winken', 'Blinken', and 'Nod'. As a result, the loop iterates three times.

Program 5-7 (simple_loop3.py)

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of numbers.
3
4  def main():
5      for name in ['Winken', 'Blinken', 'Nod']:
6          print name
7
8  # Call the main function.
9  main()
```

Program Output

```
Winken
Blinken
Nod
```

Using the `range` Function with the `for` Loop

Python provides a built-in function named `range` that simplifies the process of writing a count-controlled `for` loop. Here is an example of a `for` loop that uses the `range` function:

```
for num in range(5):
    print num
```

Notice that instead of using a list of values, we call to the `range` function passing 5 as an argument. In this statement the `range` function will generate a list of integers in the range of 0 up to (but not including) 5. This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
    print num
```

As you can see, the list contains five numbers, so the loop will iterate five times. Program 5-8 uses the `range` function with a `for` loop to display "Hello world" five times.

Program 5-8 (simple_loop4.py)

```
# This program demonstrates how the range
# function can be used with a for loop.

def main():
    # Print a message five times.
    for x in range(5):
        print 'Hello world!'

# Call the main function.
main()
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

If you pass one argument to the `range` function, as demonstrated in Program 5-8, that argument is used as the ending limit of the list. If you pass two arguments to the `range` function, the first argument is used as the starting value of the list and the second argument is used as the ending limit. Here is an example:

```
for num in range(1, 5):
    print num
```

This code will display the following:

```
1
2
3
4
```

By default, the `range` function produces a list of numbers that increase by 1 for each successive number in the list. If you pass a third argument to the `range` function, that argument is used as step ***value.*** Instead of increasing by 1, each successive number in the list will increase by the step value. Here is an example:

```
for num in range(1, 10, 2):
    print num
```

In this `for` statement, three arguments are passed to the `range` function:

- The first argument, 1, is the starting value for the list.
- The second argument, 10, is the ending limit of the list. This means that the last number in the list will be 9.

- The third argument, 2, is the step value. This means that 2 will be added to each successive number in the list.

This code will display the following:

```
1
3
5
7
9
```

Using the Target Variable Inside the Loop

In a `for` loop, the purpose of the target variable is to reference each item in a sequence of items as the loop iterates. In many situations it is helpful to use the target variable in a calculation or other task within the body of the loop. For example, suppose you need to write a program that displays the numbers 1 through 10 and their squares, in a table similar to the following:

Number	Square
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

This can be accomplished by writing a `for` loop that iterates over the values 1 through 10. During the first iteration, the target variable will be assigned the value 1, during the second iteration it will be assigned the value 2, and so forth. Because the target variable will reference the values 1 through 10 during the loop's execution, you can use it in the calculation inside the loop. Program 5-9 shows how this is done.

Program 5-9 (squares.py)

```
# This program uses a loop to display a
# table showing the numbers 1 through 10
# and their Squares.

```

```
 5  def main():
 6      # Print the table headings.
 7      print 'Number\tSquare'
 8      print '---------------'
 9
10      # Print the numbers 1 through 10
11      # and their squares.
12      for number in range(1, 11):
13          square = number**2
14          print number, '\t', square
15
16  # Call the main function.
17  main()
```

Program Output

```
Number  Square
---------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

First, take a closer look at line 7, which displays the table headings:

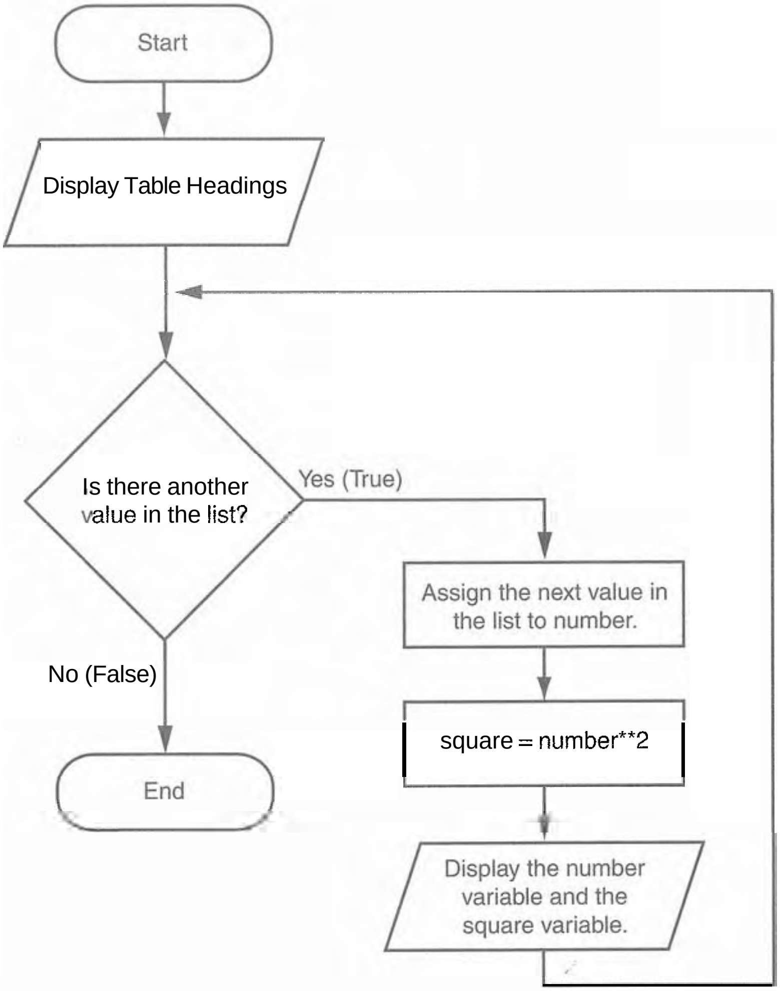
```
print 'Number\tSquare'
```

Notice that inside the string literal the `\t` escape sequence between the words Number and Square. Recall from Chapter 2 that the `\t` escape sequence is like pressing the Tab key; it causes the output cursor to move over to the next tab position. This causes the spaces that you see between the words Number and Square in the sample output.

The `for` loop that begins in line 12 uses the `range` function to produce a list containing the numbers 1 through 10. During the first iteration, `number` will reference 1, during the second iteration `number` will reference 2, and so forth, up to 10. Inside the loop, the statement in line 13 raises `number` to the power of 2 (recall from Chapter 2 that `**` is the exponent operator), and assigns the result to the `square` variable. The statement in line 14 prints the value referenced by `number`, tabs over, and then prints the value referenced by `square`. (Tabbing over with the `\t` escape sequence causes the numbers to be aligned in two columns in the output.)

Figure 5-6 shows how we might draw a flowchart for this program.

Figure 5-6 Flowchart for Program 5-9


Start
Display Table Headings
Is there another value in the list?
Yes (True)
Assign the next value in the list to number.
square = number**2
Display the number variable and the square variable.
No (False)
End

In the Spotlight:

Designing a Count-Controlled Loop with the `for` Statement

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour. She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The formula for converting kilometers per hour to miles per hour is:

$$MPH = KPH * 0.6214$$

In the formula, *MPH* is the speed in miles per hour and *KPH* is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

KPH	MPH
60	37.284
70	43.498
80	49.712
etc. . . .	
130	80.782

After thinking about this table of values, you decide that you will write a `for` loop. The list of values that the loop will iterate over will be the kilometer-per-hour speeds. In the loop you will call the `range` function like this:

```
range(60, 131, 10)
```

The first value in the list will be 60. Notice that the third argument specifies 10 as the step value. This means that the numbers in the list will be 60, 70, 80, and so forth. The second argument specifies 131 as the list's ending limit, so the last number in the list will be 130.

Inside the loop you will use the target variable to calculate a speed in miles per hour. Program 5-10 shows the program.

Program 5-10 (speed_converter.py)

```
# This program converts the speeds 60 kph
# through 130 kph (in 10 kph increments)
# to mph.

def main():
    # Print the table headings.
    print 'kph\tmph'
    print '----------------'

    # Print the speeds.
    for kph in range(60, 131, 10):
        mph = kph * 0.6214
        print kph, '\t', mph

# Call the main function.
main()
```

Program Output

```
kph       mph
----------------
60        37.284
70        43.498
80        49.712
90        55.926
100       62.14
110       68.354
120       74.568
130       80.782
```

Letting the User Control the Loop Iterations

In many cases, the programmer knows the exact number of iterations that a loop must perform. For example, recall Program 5-9, which displays a table showing the numbers 1 through 10 and their squares. When the code was written, the programmer knew that the loop had to iterate over the values 1 through 10.

Sometimes the programmer needs to let the user control the number of times that a loop iterates. For example, what if you want Program 5-9 to be a bit more versatile by allowing the user to specify the maximum value displayed by the loop? Program 5-11 shows how you can accomplish this.

Program 5-11 (user_squares.py)

```
# This program uses a loop to display a
# table of numbers and their squares.

def main():
    # Get the ending limit.
    print 'This program displays a list of numbers'
    print '(starting at 1) and their squares.'
    end = input('How high should I go? ')

    # Print the table headings.
    print 'Number\tSquare'
    print '-----------------'

    # Print the numbers and their squares.
    for number in range(1, end + 1):
        square = number**2
        print number, '\t', square

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
This program displays a list of numbers
(starting at 1) and their squares.
How high should I go? 5 [Enter]
Number    Square
------------------
1         1
2         4
3         9
4         16
5         25
```

This program asks the user to enter a value that can be used as the ending limit for the list. This value is assigned to the `end` variable in line 8. Then, the expression `end + 1` is used in line 15 as the second argument for the `range` function. (We have to add one to `end` because otherwise the list would go up to, but not include, the value entered by the user.)

Program 5-12 shows an example that allows the user to specify both the starting value and the ending limit of the list.

Program 5-12 (user_squares2.py)

```
# This program uses a loop to display a
# table of numbers and their squares.

def main():
    # Get the starting value.
    print 'This program displays a list of numbers'
    print 'and their squares.'
    start = input('Enter the starting number: ')

    # Get the ending limit.
    end = input('How high should I go? ')

    # Print the table headings.
    print
    print 'Number\tSquare'
    print '--------------'

    # Print the numbers and their squares.
    for number in range(start, end + 1):
        square = number**2
        print number, '\t', square

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
This program displays a list of numbers
and their squares.
Enter the starting number: 5 [Enter]
How high should I go? 10 [Enter]
Number      Square
--------------------
5            25
6            36
7            49
8            64
9            81
10           100
```

Generating Lists that Range from Highest to Lowest

In the examples you have seen so far, the `range` function was used to generate a list with numbers that go from lowest to highest. Alternatively, you can use the `range` function to generate lists of numbers that go from highest to lowest. Here is an example:

```
range(10, 0, -1)
```

In this function call, the starting value is 10, the list's ending limit is 0, and the step value is −1. This expression will produce the following list:

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Here is an example of a `for` loop that prints the numbers 5 down to 1:

```
for num in range(5, 0, -1):
    print num
```

Checkpoint

5.8 Rewrite the following code so it calls the `range` function instead of using the list `[0, 1, 2, 3, 41 5]`.

```
for x in [0, 1, 2, 3, 4, 5]:
    print 'I love to program!'
```

5.9 What will the following code display?

```
for number in range(6):
    print number
```

5.10 What will the following code display?

```
for number in range(2, 6):
    print number
```

5.11 What will the following code display?

```
for number in range(0, 501, 100):
    print number
```

5.12 What will the following code display?

```
for number in range(10, 5, -1):
    print number
```

5.4 Calculating a Running Total

CONCEPT: A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator.

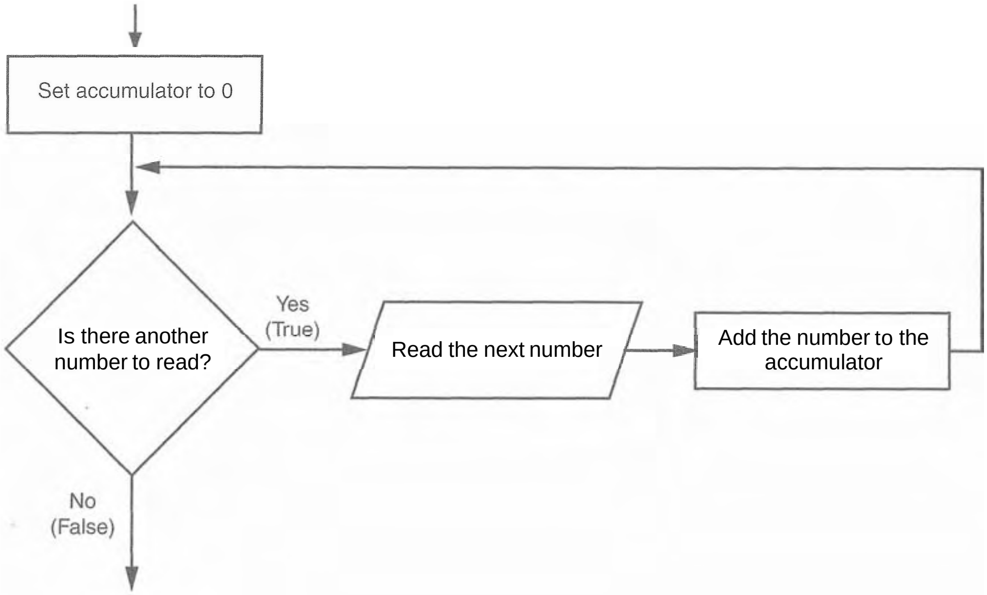
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an accumulator. It is often said that the loop keeps a running total because it accumulates the total as it reads each number in the series. Figure 5-7 shows the general logic of a loop that calculates a running total.

Figure 5-7 Logic for calculating a running total


Set accumulator to 0
Is there another number to read?
Yes (True)
Read the next number
Add the number to the accumulator
No (False)

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that calculates a running total. Program 5-13 allows the user to enter five numbers, and it displays the total of the numbers entered.

Program 5-13 (sum_numbers.py)

```
 1  # This program calculates the sum of
 2  # five numbers entered by the user.
 3
 4  def main():
 5      # Initialize an accumulator variable.
 6      total = 0.0
 7
 8      # Explain what we are doing.
 9      print 'This program calculates the sum of'
10      print 'five numbers you will enter.'
11
12      # Get five numbers and accumulate them.
13      for counter in range(5):
14          number = input('Enter a number: ')
15          total = total + number
16
17      # Display the total of the numbers.
18      print 'The total is', total
19
20  # Call the main function.
21  main()
```

Program Output (with input shown in bold)

```
This program calculates the sum of
five numbers you will enter.
Enter a number: 1 [Enter]
Enter a number: 2 [Enter]
Enter a number: 3 [Enter]
Enter a number: 4 [Enter]
Enter a number: 5 [Enter]
The total is 15.0
```

The `total` variable, created by the assignment statement in line 6, is the accumulator. Notice that it is initialized with the value 0.0. The `for` loop, in lines 13 through 15, does the work of getting the numbers from the user and calculating their total. Line 14 prompts the user to enter a number, and then assigns the input to the `number` variable. Then, the following statement in line 15 adds `number` to `total`:

```
total = total + number
```

After this statement executes, the value referenced by the `number` variable will be added to the value in the `total` variable. It's important that you understand how this statement works. First, the interpreter gets the value of the expression on the right side of the = operator, which is `total + number`. Then, that value is assigned by the = operator to the `total` variable. The effect of the statement is that the value of the `number` variable is added to the `total` variable. When the loop finishes, the `total` variable will hold the sum of all the numbers that were added to it. This value is displayed in line 18.

The Augmented Assignment Operators

Quite often, programs have assignment statements in which the variable that is on the left side of the = operator also appears on the right side of the = operator. Here is an example:

```
x = x + 1
```

On the right side of the assignment operator, 1 is added to `x`. The result is then assigned to `x`, replacing the value that `x` previously referenced. Effectively, this statement adds 1 to x. You saw another example of this type of statement in Program 5-14:

```
total = total + number
```

This statement assigns the value of `total + number` to `total`. As mentioned before, the effect of this statement is that `number` is added to the value of `total`. Here is one more example:

```
balance = balance - withdrawal
```

This statement assigns the value of the expression `balance - withdrawal` to `balance`. The effect of this statement is that `withdrawal` is subtracted from `balance`.

Table 5-1 shows other examples of statements written this way.

Table 5-1 Various assignment statements (assume `x = 6` in each statement)

Statement	What It Does	Value of `x` after the Statement
`x = x + 4`	Add 4 to `x`	10
`x = x - 3`	Subtracts 3 from `x`	3
`x = x * 10`	Multiplies `x` by 10	60
`x = x / 2`	Divides `x` by 2	3
`x = x % 4`	Assigns the remainder of `x / 4` to `x`	2

These types of operations are common in programming. For convenience, Python offers a special set of operators designed specifically for these jobs. Table 5-2 shows the augmented assignment operators.

Table 5-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5`	`x = x + 5`
	`y -= 2`	`y = y - 2`
	`z *= 10`	`z = z * 10`
`/=`	`a /= b`	`a = a / b`
`%=`	`c %= 3`	`c = c % 3`

As you can see, the augmented assignment operators do not require the programmer to type the variable name twice. The following statement:

```
total = total + number
```

could be rewritten as

```
total += number
```

Similarly, the statement

```
balance = balance - withdrawal
```

could be rewritten as

```
balance -= withdrawal;
```

Checkpoint

5.13 What is an accumulator?

5.14 Should an accumulator be initialized to any specific value? Why or why not?

5.15 What will the following code display?

```
total = 0
for count in range(1, 6):
    total = total + count
print total
```

5.16 What will the following code display?

```
numberl = 10
number2 = 5
numberl = numberl + number2
print numberl
print number2
```

5.17 Rewrite the following statements using augmented assignment operators:

a) `quantity = quantity + 1`
b) `days_left = days_left - 5`
c) `price = price * 10`
d) `price = price / 2`

5.5 Sentinels

CONCEPT: A sentinel is a special value that marks the end of a sequence of values.

Consider the following scenario: You are designing a program that will use a loop to process a long sequence of values. At the time you are designing the program, you do not know the number of values that will be in the sequence. In fact, the number of values in the sequence could be different each time the program is executed. What is the best way to design such a loop? Here are some techniques that you have seen already in this chapter, along with the disadvantages of using them when processing a long list of values:

- Simply ask the user, at the end of each loop iteration, if there is another value to process. If the sequence of values is long, however, asking this question at the end of each loop iteration might make the program cumbersome for the user.
- Ask the user at the beginning of the program how many items are in the sequence. This might also inconvenience the user, however. If the sequence is very long, and the user does not know the number of items it contains, it will require the user to count them.

When processing a long sequence of values with a loop, perhaps a better technique is to use a sentinel. A sentinel is a special value that marks the end of a sequence of items. When a program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.

For example, suppose a doctor wants a program to calculate the average weight of all her patients. The program might work like this: A loop prompts the user to enter either a patient's weight, or 0 if there are no more weights. When the program reads 0 as a weight, it interprets this as a signal that there are no more weights. The loop ends and the program displays the average weight.

A sentinel value must be distinctive enough that it will not be mistaken as a regular value in the sequence. In the example cited above, the doctor (or her medical assistant) enters 0 to signal the end of the sequence of weights. Because no patient's weight will be 0, this is a good value to use as a sentinel.

In the Spotlight:
Using a Sentinel

The county tax office calculates the annual taxes on property using the following formula:

property tax = property value × 0.0065

Every day, a clerk in the tax office gets a list of properties and has to calculate the tax for each property on the list. You have been asked to design a program that the clerk can use to perform these calculations.

In your interview with the tax clerk, you learn that each property is assigned a lot number, and all lot numbers are 1 or greater. You decide to write a loop that uses the number 0 as a sentinel value. During each loop iteration, the program will ask the clerk to enter either a property's lot number, or 0 to end. The code for the program is shown in Program 5-15.

Program 5-14 (property_tax.py)

```
# This program displays property taxes.

# TAX_FACTOR is used as a global constant
# for the tax factor.
TAX_FACTOR = 0.0065

# The main function.
def main():
```

(program continues)

Program 5-14 *(continued)*

```
    # Get the first lot number.
    print "Enter the property's lot number"
    print "or enter 0 to end."
    lot = input("Lot number: ")

    # Continue processing as long as the user
    # does not enter lot number 0.
    while lot != 0:
        # Show the tax for the property.
        show_tax()

        # Get the next lot number.
        print "Enter the next lot number or"
        print "enter 0 to end."
        lot = input("Lot number: ")

# The show_tax function gets a property's
# value and displays its tax.
def show_tax():
    # Get the property value.
    value = input("Enter the property's value: ")

    # Calculate the property's tax.
    tax = value * TAX_FACTOR

    # Display the tax.
    print "The property's tax is $%.2f." % tax

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the property's lot number
or enter 0 to end.
Lot number: 100 [Enter]
Enter the property's value: 100000.00 [Enter]
The property's tax is $650.00.
Enter the next lot number or
enter 0 to end.
Lot number: 200 [Enter]
Enter the property's value: 5000.00 [Enter]
The property's tax is $32.50.
Enter the next lot number or
enter 0 to end.
Lot number: 0 [Enter]
```

Checkpoint

5.18 What is a sentinel?

5.19 Why should you take care to choose a distinctive value as a sentinel?

5.6 Input Validation Loops

CONCEPT: Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation. Input validation is commonly done with a loop that iterates as long as an input variable references bad data.

One of the most famous sayings among computer programmers is "garbage in, garbage out." This saying, sometimes abbreviated as GIGO, refers to the fact that computers cannot tell the difference between good data and bad data. If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output. For example, look at the payroll program in Program 5-15 and notice what happens in the sample run when the user gives bad data as input.

Program 5-15 (gross_pay.py)

```
# This program displays gross pay.
def main():
    # Get the number of hours worked.
    hours = input('Enter the hours worked this week: ')

    # Get the hourly pay rate.
    pay-rate = input('Enter the hourly pay rate: ')

    # Calculate the gross pay.
    gross-pay = hours * pay-rate

    # Display the gross pay.
    print 'The gross pay is $%.2f.' % gross-pay

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the hours worked this week: 400 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $8000.00
```

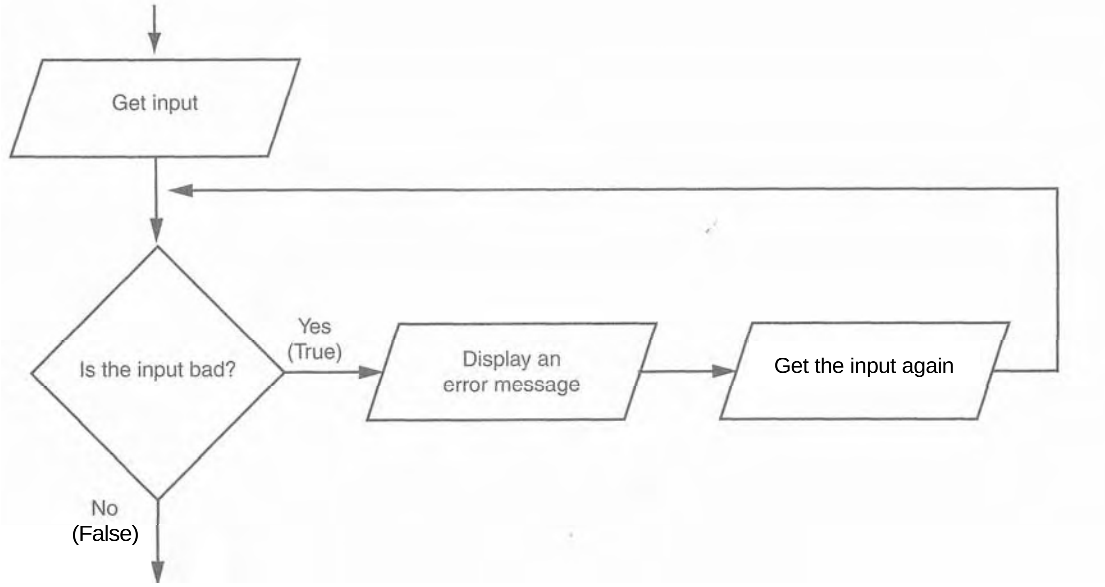
Did you spot the bad data that was provided as input? The person receiving the paycheck will be pleasantly surprised, because in the sample run the payroll clerk entered 400 as the number of hours worked. The clerk probably meant to enter 40, because there are not 400 hours in a week. The computer, however, is unaware of this fact, and the program processed the bad data just as if it were good data. Can you think of other types of input that can be given to this program that will result in bad output? One example is a negative number entered for the hours worked; another is an invalid hourly pay rate.

Sometimes stories are reported in the news about computer errors that mistakenly cause people to be charged thousands of dollars for small purchases or to receive large tax refunds that they were not entitled to. These "computer errors" are rarely caused by the computer, however; they are more commonly caused by bad data that was read into a program as input.

The integrity of a program's output is only as good as the integrity of its input. For this reason, you should design your programs in such a way that bad input is never accepted. When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data. This process is known as *input validation*.

Figure 5-8 shows a common technique for validating an item of input. In this technique, the input is read, and then a loop is executed. If the input data is bad, the loop executes its block of statements. The loop displays an error message so the user will know that the input was invalid, and then it reads the new input. The loop repeats as long as the input is bad.

Figure 5-8 Logic *containing* an *input validation loop*



Notice that the flowchart in Figure 5-8 reads input in two places: first just before the loop and then inside the loop. The first input operation—just before the loop—is called a *priming read*, and its purpose is to get the first input value that will be tested by the validation loop. If that value is invalid, the loop will perform subsequent input operations.

Let's consider an example. Suppose you are designing a program that reads a test score and you want to make sure the user does not enter a value less than 0. The following code shows how you can use an input validation loop to reject any input value that is less than 0.

```
# Get a test score.
score = input('Enter a test score: ')

# Make sure it is not less than 0.
while score < 0:
    print 'ERROR: The score cannot be negative.'
    score = input ('Enter the correct score: ')
```

This code first prompts the user to enter a test score (this is the priming read), and then the `while` loop executes. Recall that the `while` loop is a pretest loop, which means it tests the expression `score < 0` before performing an iteration. If the user entered a valid test score, this expression will be false and the loop will not iterate. If the test score is invalid, however, the expression will be true and the loop's block of statements will execute. The loop displays an error message and prompts the user to enter the correct test score. The loop will continue to iterate until the user enters a valid test score.

NOTE: An input validation loop is sometimes called an *error trap* or an *error handler.*

This code rejects only negative test scores. What if you also want to reject any test scores that are greater than 100? You can modify the input validation loop so it uses a compound Boolean expression, as shown next.

```
# Get a test score.
score = input('Enter a test score: ')

# Make sure it is not less than 0 or greater than 100.
while score < 0 or score > 100:
    print 'ERROR: The score cannot be negative'
    print 'or greater than 100.'
    score = input('Enter the correct score: ')
```

The loop in this code determines whether `score` is less than 0 or greater than 100. If either is true, an error message is displayed and the user is prompted to enter a correct score.

In the Spotlight:

Writing an Input Validation Loop

Samantha owns an import business and she calculates the retail prices of her products with the following formula:

retail price = wholesale cost × 2.5

She currently uses the program shown in Program 5-16 to calculate retail prices.

Program 5-16 (retail_no_validation.py)

```
# This program calculates retail prices.

# MARK_UP is used as a global constant for
# the markup up percentage.
MARK_UP = 2.5

# The main function
def main():
    # Variable to control the loop.
    another = 'y'

    # Process one or more items.
    while another == 'y' or another == 'Y':
        # Display an item's retail price.
        show_retail()

        # Do this again?
        another = raw_input('Do you have another item? ' + \
                            '(Enter y for yes): ')

# The show_retail function gets an item's wholesale
# cost and displays the item's retail price.
def show_retail():
    # Get the item's wholesale cost.
    wholesale = input("Enter the item's wholesale cost: ")

    # Calculate the retail price.
    retail = wholesale * MARK_UP

    # Display the retail price.
    print 'The retail price is $%.2f.' % retail

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the item's wholesale cost: 10.00 [Enter]
The retail price is $25.00.
Do you have another item? (Enter y for yes): y [Enter]
Enter the item's wholesale cost: 15.00 [Enter]
The retail price is $37.50.
Do you have another item? (Enter y for yes): y [Enter]
Enter the item's wholesale cost: 12.50 [Enter]
The retail price is $31.25.
Do you have another item? (Enter y for yes): n [Enter]
```

Samantha has encountered a problem when using the program, however. Some of the items that she sells have a wholesale cost of 50 cents, which she enters into the program as 0.50. Because the 0 key is next to the key for the negative sign, she sometimes accidentally enters a negative number. She has asked you to modify the program so it will not allow a negative number to be entered for the wholesale cost.

You decide to add an input validation loop to the `show_retail` function that rejects any negative numbers that are entered into the `wholesale` variable. Program 5-17 shows the revised program, with the new input validation code shown in lines 27 through 30.

Program 5-17 (retail_with_validation.py)

```
 1  # This program calculates retail prices.
 2
 3  # MARK_UP is used as a global constant for
 4  # the markup up percentage.
 5  MARK_UP = 2.5
 6
 7  # The main function
 8  def main():
 9      # Variable to control the loop.
10      another = 'y'
11
12      # Process one or more items.
13      while another == 'y' or another == 'Y':
14          # Display an item's retail price.
15          show_retail()
16
17          # Do this again?
18          another = raw_input('Do you have another item? ' + \
19                              '(Enter y for yes): ')
20
21  # The show_retail function gets an item's wholesale
22  # cost and displays the item's retail price.
23  def show_retail():
24      # Get the item's wholesale cost.
25      wholesale = input("Enter the item's wholesale cost: ")
26
27      # Validate the wholesale cost.
28      while wholesale < 0:
29          print 'ERROR: the cost cannot be negative.'
30          wholesale = input('Enter the correct wholesale cost: ')
31
32      # Calculate the retail price.
33      retail = wholesale * MARK_UP
34
35      # Display the retail price.
36      print 'The retail price is $%.2f.' % retail
37
```

(program continues)

Program 5-17 *(continued)*

```
# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the item's wholesale cost: -.50 [Enter]
ERROR: the cost cannot be negative.
Enter the correct wholesale cost: 0.50 [Enter]
The retail price is $1.25.
Do you have another item? (Enter y for yes): n [Enter]
```

Checkpoint

5.20 What does the phrase "garbage in, garbage out" mean?
5.21 Give a general description of the input validation process.
5.22 Describe the steps that are generally taken when an input validation loop is used to validate data.
5.23 What is a priming read? What is its purpose?
5.24 If the input that is read by the priming read is valid, how many times will the input validation loop iterate?

5.7 Nested Loops

CONCEPT: A loop that is inside another loop is called a nested loop.

A nested loop is a loop that is inside another loop. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. The hour hand, however; only makes 1 revolution for every 12 of the minute hand's revolutions. And it takes 60 revolutions of the second hand for the minute hand to make 1 revolution. This means that for every complete revolution of the hour hand, the second hand has revolved 720 times. Here is a loop that partially simulates a digital clock. It displays the seconds from 0 to 59:

```
for seconds in range(60):
    print seconds
```

We can add a `minutes` variable and nest the loop above inside another loop that cycles through 60 minutes:

```
for minutes in range(60):
    for seconds in range(60):
        print minutes, ':', seconds
```

To make the simulated clock complete, another variable and loop can be added to count the hours:

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print hours, ':', minutes, ':', seconds
```

This code's output would be:

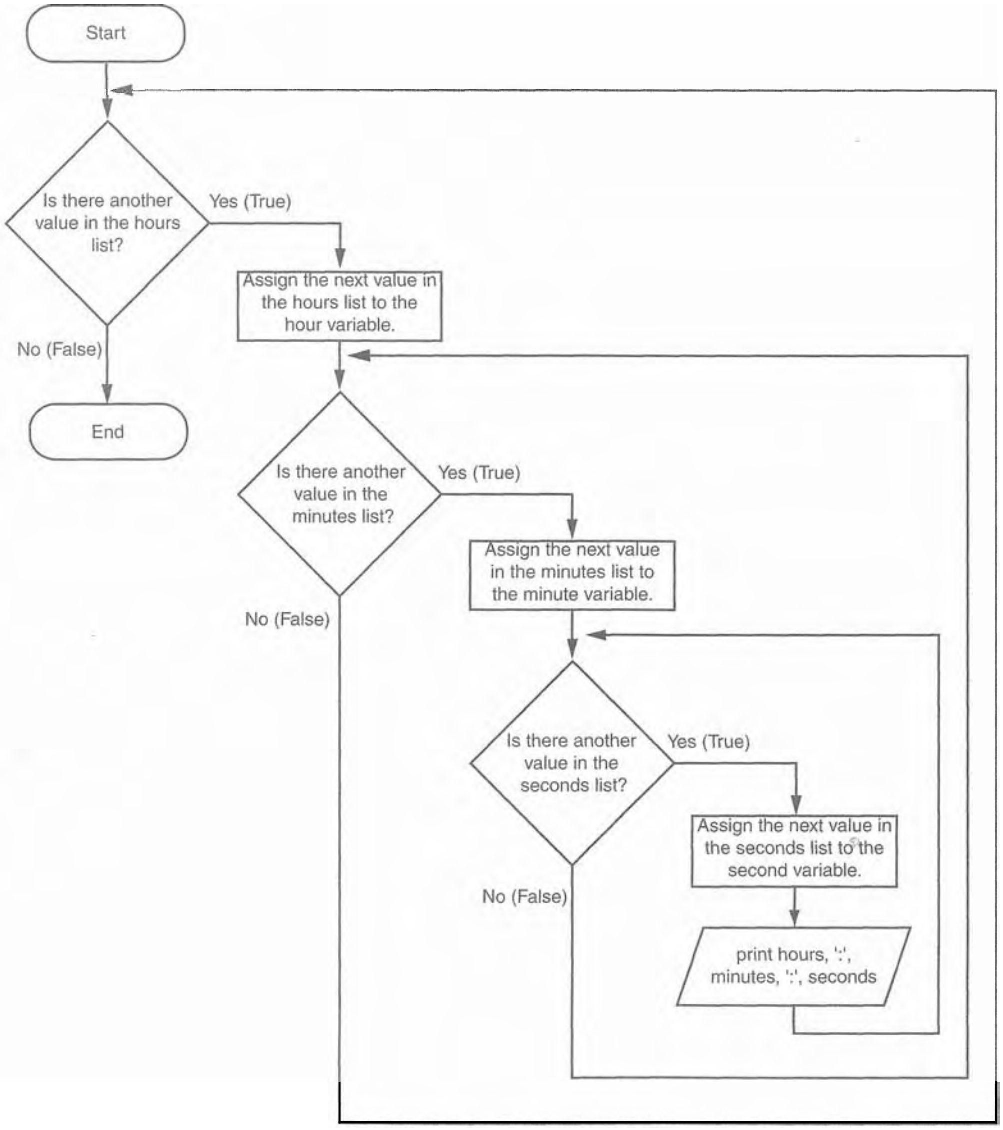
```
0:0:0
0:0:1
0:0:2
```

(The program will count through each second of 24 hours.)

```
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1440 times and the innermost loop will have iterated 86,400 times! Figure 5-9 shows a flowchart for the complete clock simulation program previously shown.

Figure 5-9 Flowchart for a clock simulator


Start
Is there another value in the hours list?
Yes (True)
Assign the next value in the hours list to the hour variable.
No (False)
End
Is there another value in the minutes list?
Yes (True)
Assign the next value in the minutes list to the minute variable.
No (False)
Is there another value in the seconds list?
Yes (True)
Assign the next value in the seconds list to the second variable.
No (False)
print hours, ':', minutes, ':', seconds

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for every single iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

Review Questions

Multiple Choice

1. A __________ -controlled loop uses a true/false condition to control the number of times that it repeats.
   a. Boolean
   b. condition
   c. decision
   d. count
2. A __________ -controlled loop repeats a specific number of times.
   a. Boolean
   b. condition
   c. decision
   d. count
3. Each repetition of a loop is known as a(n) __________.
   a. cycle
   b. revolution
   c. orbit
   d. iteration
4. The `while` loop is a __________ type of loop.
   a. pretest
   b. no-test
   c. prequalified
   d. post iterative
5. A(n) __________ loop has no way of ending and repeats until the program is interrupted.
   a. indeterminate
   b. interminable
   c. infinite
   d. timeless
6. The -= operator is an example of a(n) __________ operator.
   a. relational
   b. augmented assignment
   c. complex assignment
   d. reverse assignment

7. A(n) ________ variable keeps a running total.
   a. sentinel
   b. sum
   c. total
   d. accumulator
8. A(n) ________ is a special value that signals when there are no more items from a list of items to be processed. This value cannot be mistaken as an item from the list.
   a. sentinel
   b. flag
   c. signal
   d. accumulator
9. GIGO stands for
   a. great input, great output
   b. garbage in, garbage out
   c. GIGahertz Output
   d. GIGabyte Operation
10. The integrity of a program's output is only as good as the integrity of the program's
    a. compiler
    b. programming language
    c. input
    d. debugger
11. The input operation that appears just before a validation loop is known as the
    a. prevalidation read
    b. primordial read
    c. initialization read
    d. priming read
12. Validation loops are also known as
    a. error traps
    b. doomsday loops
    c. error avoidance loops
    d. defensive loops

True or False

1. A condition-controlled loop always repeats a specific number of times.
2. The `while` loop is a pretest loop.
3. The following statement subtracts 1 from `x`: `x = x - 1`
4. It is not necessary to initialize accumulator variables.
5. In a nested loop, the inner loop goes through all of its iterations for every single iteration of the outer loop.
6. To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.
7. The process of input validation works as follows: when the user of a program enters invalid data, the program should ask the user "Are you sure you meant to enter that?" If the user answers "yes," the program should accept the data.

Short Answer

1. What is a condition-controlled loop?
2. What is a count-controlled loop?
3. What is an infinite loop? Write the code for an infinite loop.
4. Why is it critical that accumulator variables are properly initialized?
5. What is the advantage of using a sentinel?
6. Why must the value chosen for use as a sentinel be carefully selected?
7. What does the phrase "garbage in, garbage out" mean?
8. Give a general description of the input validation process.

Algorithm Workbench

1. Write a `while loop` that lets the user enter a number. The number should be multiplied by 10, and the result assigned to a variable named `product`. The loop should iterate as long as `product` is less than 100.
2. Write a `while loop` that asks the user to enter two numbers. The numbers should be added and the sum displayed. The loop should ask the user if he or she wishes to perform the operation again. If so, the loop should repeat, otherwise it should terminate.
3. Write a `for` loop that displays the following set of numbers:

   ```
   0, 10, 20, 30, 40, 50 . . . 1000
   ```
4. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
5. Write a loop that calculates the total of the following series of numbers:

   $$\frac{1}{30} + \frac{2}{29} + \frac{3}{28} + \ldots \frac{30}{1}$$
6. Rewrite the following statements using augmented assignment operators.
   a. `x = x + 1`
   b. `x = x * 2`
   c. `x = x / 10`
   d. `x = x - 100`
7. Write a set of nested loops that display 10 rows of # characters. There should be 15 # characters in each row.
8. Write code that prompts the user to enter a positive nonzero number and validates the input.
9. Write code that prompts the user to enter a number in the range of 1 through 100 and validates the input.

Programming Exercises

1. Bug Collector

A bug collector collects bugs every day for seven days. Write a program that keeps a running total of the number of bugs collected during the seven days. The loop should ask for the number of bugs collected for each day, and when the loop is finished, the program should display the total number of bugs collected.

2. Calories Burned

Running on a particular treadmill you burn 3.9 calories per minute. Write a program that uses a loop to display the number of calories burned after 10, 15, 20, 25, and 30 minutes.

3. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month, and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

4. Distance Traveled

The distance a vehicle travels can be calculated as follows:

$$distance = speed \times time$$

For example, if a train travels 40 miles per hour for three hours, the distance traveled is 120 miles. Write a program that asks the user for the speed of a vehicle (in miles per hour) and the number of hours it has traveled. It should then use a loop to display the distance the vehicle has traveled for each hour of that time period. Here is an example of the desired output:

```
What is the speed of the vehicle in mph? 40 [Enter]
How many hours has it traveled? 3 [Enter]
Hour            Distance Traveled
--------------------------------
 1                    40
 2                    80
 3                   120
```

5. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. The program should first ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate twelve times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month. After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

6. Celsius to Fahrenheit Table

Write a program that displays a table of the Celsius temperatures 0 through 20 and their Fahrenheit equivalents. The formula for converting a temperature from Celsius to Fahrenheit is

$$F = \frac{9}{5}C + 32$$

where F is the Fahrenheit temperature and C is the Celsius temperature. Your program must use a loop to display the table.

7. Pennies for Pay

Write a program that calculates the amount of money a person would earn over a period of time if his or her salary is one penny the first day, two pennies the second day, and continues to double each day. The program should ask the user for the number of days. Display a table showing what the salary was for each day, and then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.

8. Sum of Numbers

Write a program with a loop that asks the user to enter a series of positive numbers. The user should enter a negative number to signal the end of the series. After all the positive numbers have been entered, the program should display their sum.

CHAPTER

6 Value-Returning Functions and Modules

TOPICS

6.1 Introduction to Value-Returning Functions: Generating Random Numbers
6.2 Writing Your Own Functions
6.3 The `math` Module
6.4 Storing Functions in Modules

6.1 Introduction to Value-Returning Functions: Generating Random Numbers

CONCEPT: A value-returning function is a function that returns a value back to the part of the program that called it. Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks. These libraries typically contain a function that generates random numbers.

In Chapter 3 you learned about simple functions. A simple function is a group of statements that exist within a program for the purpose of performing a specific task. When you need the function to perform its task, you call the function. This causes the statements inside the function to execute. When the function is finished, control of the program returns to the statement appearing immediately after the function call.

A *value-returning function* is a special type of function. It is like a simple function in the following ways.

- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

When a value-returning function finishes, however, it returns a value back to the part of the program that called it. The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

Standard Library Functions

Python, as well as most other programming languages, comes with a standard library of functions that have already been written for you. These functions, known as library functions, make a programmer's job easier because they perform many of the tasks that programmers commonly need to perform. In fact, you have already used several of Python's library functions. Some of the functions that you have used are `input`, `raw_input`, and `range`. Python has many other library functions. Although we won't cover them all in this book, we will discuss library functions that perform fundamental operations.

Some of Python's library functions are built into the Python interpreter. If you want to use one of these built-in functions in a program, you simply call the function. This is the case with the `input`, `raw_input`, `range`, and other functions that you have already learned about. Many of the functions in the standard library, however, are stored in files that are known as modules. These modules, which are copied to your computer when you install Python, help organize the standard library functions. For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on. In order to call a function that is stored in a module, you have to write an `import` statement at the top of your program. An `import` statement tells the interpreter the name of the module that contains the function.

Because you do not see the internal workings of library functions, many programmers think of them as black boxes. The term "black box" is used to describe any mechanism that accepts input, performs some operation (that cannot be seen) using the input, and produces output. Figure 6-1 illustrates this idea.

Figure 6-1 A library function viewed as a black box

Input
Library Function
Output

This section demonstrates how functions work by looking at standard library functions that generate random numbers, and some interesting programs that can be written with them. Then you will learn to write your own value-returning functions and how to create your own modules. The last section in this chapter comes back to the topic of library functions and looks at several other useful functions in the Python standard library.

Generating Random Numbers

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will

behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.

- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

Python provides several library functions for working with random numbers. These functions are stored in a module named random in the standard library. To use any of these functions you first need to write this `import` statement at the top of your program:

```
import random
```

This statement causes the interpreter to load the contents of the random module into memory. This makes all of the functions in the random module available to your program[1]

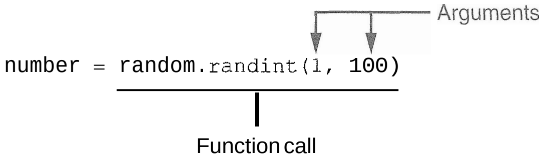
The first random-number generating function that we will discuss is named `randint`. Because the `randint` function is in the random module, we will need to use *dot* notation to refer to it in our program. In dot notation, the function's name is random. `randint`. On the left side of the dot (period)is the name of the module, and on the right side of the dot is the name of the function.

The following statement shows an example of how you might call the `randint` function.

```
number = random.randint(1, 100)
```

The part of the statement that reads random. `randint(1, 100)` is a call to the `randint` function. Notice that two arguments appear inside the parentheses: 1 and 100. These arguments tell the function to give an integer random number in the range of 1 through 100. (The values 1 and 100 are included in the range.) Figure 6-2 illustrates this part of the statement.

Figure 6-2 A statement that calls the random function



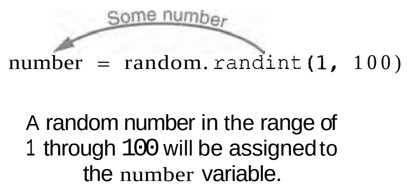
Notice that the call to the `randint` function appears on the right side of an = operator. When the function is called, it will generate a random number in the range of 1 through 100 and then *return* that number. The number that is returned will be assigned to the number variable, as shown in Figure 6-3.

Program 6-1 shows a complete program that uses the `randint` function. The statement in line 2 generates a random number in the range of 1 through 10 and assigns it to the number variable. (The program output shows that the number 7 was generated, but this value is arbitrary. If this were an actual program, it could display any number from 1 to 10.)

---

[1] There are several ways to write an `import` statement in Python, and each variation works a little differently. Many Python programmers agree that the preferred way to import a module is the way shown in this book.

Figure 6-3 The random function returns a value


Some number
number = random.randint(1, 100)
A random number in the range of 1 through 100 will be assigned to the number variable.

Program 6-1 (random_numbers.py)

```
# This program displays a random number
# in the range of 1 through 10.
import random

def main():
    # Get a random number.
    number = random.randint(1, 10)
    # Display the number.
    print 'The number is', number

# Call the main function.
main()
```

Program Output

```
The number is 7
```

Program 6-2 shows another example. This program uses a **`for`** loop that iterates five times. Inside the loop, the statement in line 8 calls the `randint` function to generate a random number in the range of 1 through 100.

Program 6-2 (random-numbers2.py)

```
 1  # This program displays five random
 2  # numbers in the range of 1 through 100.
 3  import random
 4
 5  def main():
 6      for count in range(5):
 7          # Get a random number.
 8          number = random.randint(1, 100)
 9          # Display the number.
10          print number
11
12  # Call the main function.
13  main()
```

Program Output

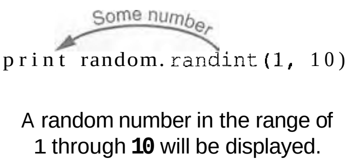
```
89
7
16
41
12
```

Both Programs 6-1 and 6-2 call the `randint` function and assign its return value to the number variable. If you just want to display a random number, it is not necessary to assign the random number to a variable. You can send the `random` function's return value directly to the `print` statement, as shown here:

```
print random.randint(1, 10)
```

When this statement executes, the `randint` function is called. The function generates a random number in the range of 1 through 10. That value is returned and then sent to the `print` statement. As a result, a random number in the range of 1 through 10 will be displayed. Figure 6-4 illustrates this.

Figure 6-4 Displaying a random number


Some number
print random.randint(1, 10)
A random number in the range of
1 through 10 will be displayed.

Program 6-3 shows how you could simplify Program 6-2. This program also displays five random numbers, but this program does not use a variable to hold those numbers. The `randint` function's return value is sent directly to the `print` statement in line 7.

Program 6-3 (random-numbers3.py)

```
 1  # This program displays five random
 2  # numbers in the range of 1 through 100.
 3  import random
 4
 5  def main():
 6      for count in range(5):
 7          print random.randint(1, 100)
 8
 9  # Call the main function.
10  main()
```

Program Output

```
89
7
16
41
12
```

In the Spotlight:

Using Random Numbers

Dr. Kimura teaches an introductory statistics class, and has asked you to write a program that he can use in class to simulate the rolling of dice. The program should randomly generate two numbers in the range of 1 through 6 and display them. In your interview with Dr. Kimura, you learn that he would like to use the program to simulate several rolls of the dice, one after the other. Here is the pseudocode for the program:

```
While the user wants to roll the dice:
    Display a random number in the range of 1 through 6
    Display another random number in the range of 1 through 6
    Ask the user if he or she wants to roll the dice again
```

You will `write` a while loop that simulates one roll of the dice, and then asks the user if another roll should be performed. As long as the user answers "y" for yes, the loop will repeat. Program 6-4 shows the program.

Program 6-4 (dice.py)

```
# This program simulates the rolling of dice.
import random

def main():
    # Create a variable to control the loop.
    again = 'y'

    # Simulate rolling the dice.
    while again == 'y' or again == 'Y':
        print 'Rolling the dice...'
        print 'Their values are:'
        print random.randint(1, 6)
        print random.randint(1, 6)

        # Do another roll of the dice?
        again = raw_input('Roll them again? (y = yes): ')

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Rolling the dice...
Their values are:
3
1
Roll them again? (y = yes): y [Enter]
Rolling the dice...
```

```
Their values are:
1
1
Roll them again? (y = yes): y [Enter]
Rolling the dice...
Their values are:
5
6
Roll them again? (y = yes): y [Enter]
```

The `randint` function returns an integer value, so you can write a call to the function anywhere that you can write an integer value. You have already seen examples where the function's return value is assigned to a variable and where the function's return value is sent to the `print` statement. To further illustrate the point, here is a statement that uses the `randint` function in a math expression:

```
x = random.randint(1, 10) * 2
```

In this statement, a random number in the range of 1 through 10 is generated and then multiplied by 2. The result is a random integer from 2 to 20 assigned to the x variable. You can also test the return value of the function with an `if` statement, as demonstrated in the following In the Spotlight section.

In the Spotlight:

Using Random Numbers to Represent Other Values

Dr. Kimura was so happy with the dice rolling simulator that you wrote for him, he has asked you to write one more program. He would like a program that he can use to simulate ten coin tosses, one after the other. Each time the program simulates a coin toss, it should randomly display either “Heads” or “Tails”.

You decide that you can simulate the tossing of a coin by randomly generating a number in the range of 1 through 2. You will write an `if` statement that displays “Heads” if the random number is 1, or “Tails” otherwise. Here is the pseudocode:

```
Repeat 10 times:
        If a random number in the range of 1 through 2 equals 1 then:
           Display ‘Heads’
        Else:
           Display ‘Tails’
```

Because the program should simulate 10 tosses of a coin you decide to use a `for` loop. The program is shown in Program 6-5.

Program 6-5 (coin_toss.py)

```
# This program simulates 10 tosses of a coin.
import random

def main():
    for toss in range(10):
        # Simulate the coin toss.
        if random.randint(1, 2) == 1:
            print 'Heads'
        else:
            print 'Tails'

# Call the main function.
main()
```

Program Output

```
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Tails
Heads
Tails
```

The `randrange`, `random`, and `uniform` Functions

The standard library's `random` module contains numerous functions for working with random numbers. In addition to the `randint` function, you might find the `randrange`, `random`, and `uniform` functions useful. (To use any of these functions you need to write `import random` at the top of your program.)

If you remember how to use the `range` function (which we discussed in Chapter 5) then you will immediately be comfortable with the `randrange` function. The `randrange` function takes the same arguments as the `range` function. The difference is that the `randrange` function does not return a list of values. Instead, it returns a randomly selected value from a sequence of values. For example, the following statement assigns a random number in the range of 0 through 9 to the `number` variable:

```
number = random.randrange(10)
```

The argument, in this case 10, specifies the ending limit of the sequence of values. The function will return a randomly-selected number from the sequence of values 0 up to, but not including, the ending limit. The following statement specifies both a starting value and an ending limit for the sequence:

```
number = random.randrange(5, 10)
```

When this statement executes, a random number in the range of 5 through 9 will be assigned to number. The following statement specifies a starting value, an ending limit, and a step value:

```
number = random.randrange(0, 101, 10)
```

In this statement the `randrange` function returns a randomly selected value from the following sequence of numbers:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Both the `randint` and the `randrange` functions return an integer number. The `random` function returns, however, returns a random floating-point number. You do not pass any arguments to the `random` function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0). Here is an example:

```
number = random.random()
```

The `uniform` function also returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

```
number = random.uniform(l.0, 10.0)
```

In this statement the `uniform` function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the `number` variable.

Checkpoint

6.1 How does a value-returning function differ from the simple functions we discussed in Chapter 3?

6.2 What is a library function?

6.3 Why are library functions like "black boxes"?

6.4 What does the following statement do?

```
x = random.randint(1, 100)
```

6.5 What does the following statement do?

```
print random.randint(1, 20)
```

6.6 What does the following statement do?

```
print random.randrange(10, 20)
```

6.7 What does the following statement do?

```
print random.random()
```

6.8 What does the following statement do?

```
print random.uniform(0.1, 0.5)
```

6.2 Writing Your Own Value-Returning Functions

CONCEPT: A value-returning function has a return statement that returns a value back to the part of the program that called it.

You write a value-returning function in the same way that you write a simple function, with one exception: a value-returning function must have a `return` statement. Here is the general format of a value-returning function definition in Python:

```
def function-name():
    statement
    statement
    etc.
    return expression
```

One of the statements in the function must be a `return` statement, which takes the following form:

```
return expression
```

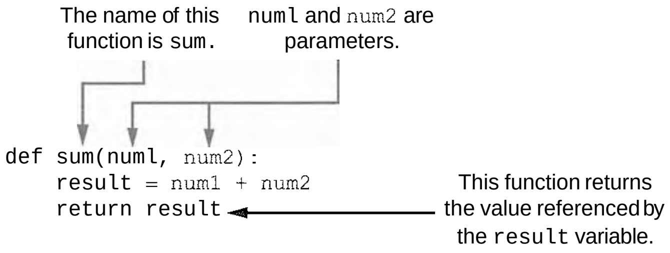
The value of the *`expression`* that follows the key word `return` will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value (such as a math expression).

Here is a simple example of a value-returning function:

```
def sum(num1, num2):
    result = numl + num2
    return result
```

Figure 6-5 illustrates various parts of the function.

Figure 6-5 Pasts of the function


The name of this function is sum.
numl and num2 are parameters.
This function returns the value referenced by the result variable.

```
def sum(numl, num2):
    result = num1 + num2
    return result
```

The purpose of this function is to accept two integer values as arguments and return their sum. Let's take a closer look at how it works. The first statement in the function's block assigns the value of `num1 + num2` to the `result` variable. Next, the `return` statement executes, which causes the function to end execution and sends the value referenced by the `result` variable back to the part of the program that called the function. Program 6-6 demonstrates the function.

Program 6-6 (total_ages.py)

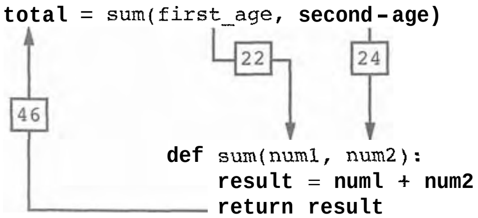
```
 1  # This program uses the return value of a function.
 2
 3  def main():
 4      # Get the user's age.
 5      first_age = input('Enter your age: ')
 6
 7      # Get the user's best friend's age.
 8      second-age = input("Enter your best friend's age: ")
 9
10      # Get the sum of both ages.
11      total = sum(first_age, second-age)
12
13      # Display the total age.
14      print 'Together you are', total, 'years old.'
15
16  # The sum function accepts two numeric arguments and
17  # returns the sum of those arguments.
18  def sum(num1, num2):
19      result = numl + num2
20      return result
21
22  # Call the main function.
23  main()
```

Program Output (with input shown in bold)

```
Ent[illegible]r age: 22 [Enter]
Ent[illegible]r best friend's age: 24 [Enter]
Togetner you are 46 years old.
```

In the **main** function, the program gets two values from the user and stores them in the **first_age** and **second-age** variables. The statement in line 11 calls the **sum** function, passing **first-age** and **second-age** as arguments. The value that is returned from the **sum** function is assigned to the **total** variable. In this case, the function will return 46. Figure 6-6 shows how the arguments are passed into the function, and how a value is returned back from the function.

Figure 6-6 Arguments ate passed to the **sum** function and a value is returned



```
total = sum(first_age, second-age)

def sum(numl, num2):
    result = numl + num2
    return result
```

Making the Most of the `return` Statement

Look again at the sum function presented in Program 6-6:

```
def sum(numl, num2):
    result = numl + num2
    return result
```

Notice that two things happen inside this function: (1) the value of the expression `num1 + num2` is assigned to the `result` variable, and (2) the value of the `result` variable is returned. Although this function does what it sets out to do, it can be simplified. Because the `return` statement can return the value of an expression, you can eliminate the `result` variable and rewrite the function as:

```
def sum( numl, num2):
    return numl + num2
```

This version of the function does not store the value of `num1 + num2` in a variable. Instead, it takes advantage of the fact that the `return` statement can return the value of an expression. This version of the function does the same thing as the previous version, but in only one step.

How to Use Value-Returning Functions

Value-returning functions provide many of the same benefits as simple functions: they simplify code, reduce duplication, enhance your ability to test code, increase the speed of development, and ease the facilitation of teamwork.

Because value-returning functions return a value, they can be useful in specific situations. For example, you can use a value-returning function to prompt the user for input, and then it can return the value entered by the user. Suppose you've been asked to design a program that calculates the sale price of an item in a retail business. To do that, the program would need to get the item's regular price from the user. Here is a function you could define for that purpose:

```
def get-regular-price():
    price = input("Enter the item's regular price: ")
    return price
```

Then, elsewhere in the program, you could call that function, as shown here:

```
# Get the item's regular price.
reg-price = get-regular-price()
```

When this statement executes, the `get_regular_price` function is called, which gets a value from the user and returns it. That value is then assigned to the `reg_price` variable.

You can also use functions to simplify complex mathematical expressions. For example, calculating the sale price of an item seems like it would be a simple task: you calculate the discount and subtract it from the regular price. In a program, however, a statement that performs this calculation is not that straightforward, as shown in the following example. (Assume `DISCOUNT_PERCENTAGE` is a global constant that is defined in the program, and it specifies the percentage of the discount.)

```
sale-price = reg-price - (reg-price * DISCOUNT_PERCENTAGE)
```

This statement isn't easy to understand because it performs so many steps: it calculates the discount amount, subtracts that value from `reg—price`, and assigns the result to **`sale_price.`** You could simplify the statement by breaking out part of the math expression and placing it in a function. Here is a function named `discount` that accepts an item's price as an argument and returns the amount of the discount:

```
def discount(price):
    return price * DISCOUNT_PERCENTAGE
```

You could then call the function in your calculation:

```
sale-price = reg-price - discount(reg_price)
```

This statement is easier to read than the one previously shown, and it is clearer that the discount is being subtracted from the regular price. Program 6-7 shows the complete sale price calculating program using the functions just described.

Program 6-7 (sale_price.py)

```
# This program calculates a retail item's
# sale price.

# DISCOUNT-PERCENTAGE is used as a global
# constant for the discount percentage.
DISCOUNT-PERCENTAGE = 0.20

# The main function.
def main():
    # Get the item's regular price.
    reg-price = get-regular-price()

    # Calculate the sale price.
    sale-price = reg-price – discount(reg_price)

    # Display the sale price.
    print 'The sale price is $%.2f.' % sale-price

# The get-regular-price function prompts the
# user to enter an item's regular price and it
# returns that value.
def get-regular-price():
    price = input("Enter the item's regular price: ")
    return price

# The discount function accepts an item's price
# as an argument and returns the amount of the
# discount, specified by DISCOUNT-PERCENTAGE.
def discount(price):
```

(program continues)

Program 6-7 *(continued)*

```
    return price * DISCOUNT_PERCENTAGE

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the item's regular price: 100.00 [Enter]
The sale price is $80.00
```

Using IPO Charts

An IPO chart is a simple but effective tool that programmers sometimes use for designing and documenting functions. IPO stands for *input*, *processing*, and *output*, and an *IPO chart* describes the input, processing, and output of a function. These items are usually laid out in columns: the input column shows a description of the data that is passed to the function as arguments, the processing column shows a description of the process that the function performs, and the output column describes the data that is returned from the function. For example, Figure 6-7 shows IPO charts for the `get_regular_price` and `discount` functions that you saw in Program 6-7.

Figure 6-7 IPO charts for the `getRegularPrice` and **`discount`** functions

IPO Chart for the `get_regular_price` Function		
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price

IPO Chart for the `discount` Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant `DISCOUNT_PERCENTAGE`.	The item's discount

Notice that the IPO charts provide only brief descriptions of a function's input, processing, and output, but do not show the specific steps taken in a function. In many cases, however, IPO charts include sufficient information so that they can be used instead of a flowchart. The decision of whether to use an IPO chart, a flowchart, or both is often left to the programmer's personal preference.

In the Spotlight:
Modularizing with Functions

Hal owns a business named Make Your Own Music, which sells guitars, drums, banjos, synthesizers, and many other musical instruments. Hal's sales staff works strictly on commission. At the end of the month, each salesperson's commission is calculated according to Table 6-1.

Table 6-1 Sales commission rates

Sales This Month	Commission Rate
Less than $10,000	10%
$10,000–14,999	12%
$15,000–17,999	14%
$18,000–21,999	16%
$22,000 or more	18%

For example, a salesperson with $16,000 in monthly sales will earn a 14 percent commission ($2240). Another salesperson with $18,000 in monthly sales will earn a 16 percent commission ($2880). A person with $30,000 in sales will earn an 18 percent commission ($5400).

Because the staff gets paid once per month, Hal allows each employee to take up to $2000 per month in advance. When sales commissions are calculated, the amount of each employee's advanced pay is subtracted from the commission. If any salesperson's commissions are less than the amount of their advance, they must reimburse Hal for the difference. To calculate a salesperson's monthly pay, Hal uses the following formula:

$$\textit{pay} = \textit{sales} \times \textit{commission rate} - \textit{advanced pay}$$

Hal has asked you to write a program that makes this calculation for him. The following general algorithm outlines the steps the program must take.

1. *Get the salesperson's monthly sales.*
2. *Get the amount of advanced pay.*
3. *Use the amount of monthly sales to determine the commission rate.*
4. *Calculate the salesperson's pay using the formula previously shown. If the amount is negative, indicate that the salesperson must reimburse the company.*

Program 6-8 shows the code, which is written using several functions. Rather than presenting the entire program at once, let's first examine the `main` function and then each function separately. Here is the `main` function:

Program 6-8 (commission_rate.py) main function

```
 1  # This program calculates a salesperson's pay
 2  # at Make Your Own Music.
 3  def main():
 4      # Get the amount of sales.
 5      sales = get_sales()
 6
 7      # Get the amount of advanced pay.
 8      advanced_pay = get_advanced_pay()
 9
10      # Determine the commission rate.
11      comm_rate = determine_comm_rate(sales)
12
13      # Calculate the pay.
14      pay = sales * comm_rate - advanced_pay
15
16      # Display the amount of pay.
17      print 'The pay is $%.2f.' % pay
18
19      # Determine whether the pay is negative.
20      if pay < 0:
21          print 'The Salesperson must reimburse'
22          print 'the company.'
23
```

Line 5 calls the `get_sales` function, which gets the amount of sales from the user and returns that value. The value that is returned from the function is assigned to the `sales` variable. Line 8 calls the `get_advanced_pay` function, which gets the amount of advanced pay from the user and returns that value. The value that is returned from the function is assigned to the `advanced_pay` variable.

Line 11 calls the `determine_comm_rate` function, passing `sales` as an argument. This function returns the rate of commission for the amount of sales. That value is assigned to the `comm_rate` variable. Line 14 calculates the amount of pay, and then line 17 displays that amount. The `if` statement in lines 20 through 22 determines whether the pay is negative, and if so, displays a message indicating that the salesperson must reimburse the company. The `get_sales` function definition is next.

Program 6-8 (commission_rate.py) get_sales function

```
24  # The get_sales function gets a salesperson's
25  # monthly sales from the user and returns that value.
26  def get_sales():
```

```
27         # Get the amount of monthly sales.
28         monthly_sales = input('Enter the monthly sales: ')
29 
30         # Return the amount entered.
31         return monthly_sales
32 
```

The purpose of the `get_sales` function is to prompt the user to enter the amount of sales for a salesperson and return that amount. Line 28 prompts the user to enter the sales, and stores the user's input in the `monthly_sales` variable. Line 31 returns the amount in the `monthly_sales` variable. Next is the definition of the `get_advanced_pay` function.

Program 6-8 (commission_rate.py) get_advanced_pay function

```
33     # The get_advanced_pay function gets the amount of
34     # advanced pay given to the salesperson and returns
35     # that amount.
36     def get_advanced_pay():
37         # Get the amount of advanced pay.
38         print 'Enter the amount of advanced pay, or'
39         print 'enter 0 if no advanced pay was given.'
40         advanced = input('Advanced pay: ')
41 
42         # Return the amount entered.
43         return advanced
44 
```

The purpose of the `get_advanced_pay` function is to prompt the user to enter the amount of advanced pay for a salesperson and return that amount. Lines 38 and 39 tell the user to enter the amount of advanced pay (or 0 if none was given). Line 40 gets the user's input and stores it in the `advanced` variable. Line 43 returns the amount in the `advanced` variable. Defining the `determine_comm_rate` function comes next.

Program 6-8 (commission_rate.py) determine_comm_rate function

```
45     # The determine_comm_rate function accepts the
46     # amount of sales as an argument and returns the
47     # applicable commission rate.
48     def determine_comm_rate(sales):
49         # Determine the commission rate.
50         if sales < 10000.00:
51             rate = 0.10
52         elif sales >= 10000 and sales <= 14999.99:
53             rate = 0.12
54         elif sales >= 15000 and sales <= 17999.99:
55             rate = 0.14
```

(program continues)

Program 6-8 *(continued)*

```
56      elif sales >= 18000 and sales <= 21999.99:
57          rate = 0.16
58      else:
59          rate = 0.18
60
61      # Return the commission rate.
62      return rate
63
```

The `determine_comm_rate` function accepts the amount of sales as an argument, and it returns the applicable commission rate for that amount of sales. The `if-elif-else` statement in lines 50 through 59 tests the `sales` parameter and assigns the correct value to the local `rate` variable. Line 62 returns the value in the local `rate` variable.

Program Output (with input shown in bold)

```
Enter the monthly sales: 14650.00 [Enter]
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 1000.00 [Enter]
The pay is $758.00.
```

Program Output (with input shown in bold)

```
Enter the monthly sales: 9000.00 [Enter]
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 0 [Enter]
The pay is $900.00.
```

Program Output (with input shown in bold)

```
Enter the monthly sales: 12000.00 [Enter]
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 2000.00 [Enter]
The pay is $-560.00.
The salesperson must reimburse
the company.
```

Returning Strings

So far you've seen examples of functions that return numbers. You can also write functions that return strings. For example, the following function prompts the user to enter his or her name, and then returns the string that the user entered.

```
def get-name():
    # Get the user's name.
    name = raw-input('Enter your name: ')
    # Return the name.
    return name
```

Returning Boolean Values

Python allows you to write Boolean functions, which return either `True` or `False`. You can use a Boolean function to test a condition, and then return either `True` or `False` to indicate whether the condition exists. Boolean functions are useful for simplifying complex conditions that are tested in decision and repetition structures.

For example, suppose you are designing a program that will ask the user to enter a number, and then determine whether that number is even or odd. The following code shows how you can make that determination.

```
number = input('Enter a number: ')
if (number % 2) == 0:
    print 'The number is even.'
else:
    print 'The number is odd.'
```

Let's take a closer look at the Boolean expression being tested by this `if-else` statement:

```
(number % 2) == 0
```

This expression uses the `%` operator, which was introduced in Chapter 2. This is called the remainder operator. It divides two numbers and returns the remainder of the division. So this code is saying, "If the remainder of `number` divided by 2 is equal to 0, then display a message indicating the number is even, or else display a message indicating the number is odd."

Because dividing an even number by 2 will always give a remainder of 0, this logic will work. The code would be easier to understand, however, if you could somehow rewrite it to say, "If the number is even, then display a message indicating it is even, or else display a message indicating it is odd." As it turns out, this can be done with a Boolean function. In this example, you could write a Boolean function named `is_even` that accepts a number as an argument and returns `True` if the number is even, or `False` otherwise. The following is the code for such a function.

```
def is_even(number):
    # Determine whether number is even. If it is,
    # set status to true. Otherwise, set status
    # to false.
    if (number % 2) == 0:
        status = True
    else:
        status = False
    # Return the value of the status variable.
    return status
```

Then you can rewrite the `if-else` statement so it calls the `is_even` function to determine whether `number` is even:

```
number = input('Enter a number: ')
if is_even(number):
    print 'The number is even.'
else:
    print 'The number is odd.'
```

Not only is this logic easier to understand, but now you have a function that you can call in the program anytime you need to test a number to determine whether it is even.

Using Boolean Functions in Validation Code

You can also use Boolean functions to simplify complex input validation code. For instance, suppose you are writing a program that prompts the user to enter a product model number and should only accept the values 100,200, and 300. You could design the input algorithm as follows:

```
# Get the model number.
model = input('Enter the model number: ')

# Validate the model number.
while model != 100 and model != 200 and model != 300:
    print 'The valid model numbers are 100, 200 and 300.'
    model = input('Enter a valid model number: ')
```

The validation loop uses a long compound Boolean expression that will iterate as long as `model` does not equal 100 *and* `model` does not equal 200 *and* `model` does not equal 300. Although this logic will work, you can simplify the validation loop by writing a Boolean function to test the `model` variable and then calling that function in the loop. For example, suppose you pass the `model` variable to a function you write named `is_invalid`. The function returns `True` if `model` is invalid, or `False` otherwise. You could rewrite the validation loop as follows:

```
# Validate the model number.
while is_invalid(model):
    print 'The valid model numbers are 100, 200 and 300.'
    model = input('Enter a valid model number: ')
```

This makes the loop easier to read. It is evident now that the loop iterates as long as `model` is invalid. The following code shows how you might write the `is_invalid` function. It accepts a model number as an argument, and if the argument is not 100 and the argument is not 200 and the argument is not 300, the function returns `True` to indicate that it is invalid. Otherwise, the function returns `False`.

```
def is_invalid(mod_num):
    if mod-num != 100 and mod-num != 200 and mod-num != 300:
        status = True
    else:
        status = False
    return status
```

Returning Multiple Values

The examples of value-returning functions that we have looked at so far return a single value. In Python, however, you are not limited to returning only one value. You can specify multiple expressions separated by commas after the `return` statement, as shown in this general format:

```
return expression1, expression2, etc.
```

As an example, look at the following definition for a function named `get_name`. The function prompts the user to enter his or her first and last names. These names are stored in two local variables: `first` and `last`. The `return` statement returns both of the variables.

```
def get-name():
    # Get the user's first and last names.
    first = raw-input('Enter your first name: ')
    last = raw-input('Enter your last name: ')

    # Return both names.
    return first, last
```

When you call this function in an assignment statement, you need to use two variables on the left side of the = operator. Here is an example:

```
first-name, last-name = get-name()
```

The values listed in the `return` statement are assigned, in the order that they appear, to the variables on the left side of the = operator. After this statement executes, the value of the `first` variable will be assigned to `first_name` and the value of the `last` variable will be assigned to `last_name`. Note that the number of variables on the left side of the = operator must match the number of values returned by the function. Otherwise an error will occur.

Checkpoint

6.9 What is the purpose of the `return` statement in a function?

6.10 Look at the following function definition:

```
def do_something(number):
    return number * 2
```

a. What is the name of the function?

b. What does the function do?

c. Given the function definition, what will the following statement display?

```
print do_something(10)
```

6.11 What is a Boolean function?

6.3 The `math` Module

CONCEPT: The Python standard library's **`math`** module contains numerous functions that can be used in mathematical calculations.

The `math` module in the Python standard library contains several functions that are useful for performing mathematical operations. Table 6-2 lists many of the functions in the `math` module. These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. For example, one of the functions is named `sqrt`. The `sqrt` function accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```
result = math.sqrt(16)
```

This statement calls the `sqrt` function, passing 16 as an argument. The function returns the square root of 16, which is then assigned to the `result` variable. Program 6-9 demonstrates the `sqrt` function. Notice the `import math` statement in line 2. You need to write this in any program that uses the `math` module.

Program 6-9 (square_root.py)

```
 1  # This program demonstrates the sqrt function.
 2  import math
 3
 4  def main():
 5      # Get a number.
 6      number = input('Enter a number: ')
 7
 8      # Get the square root of the number.
 9      square-root = math.sqrt(number)
10
11      # Display the square root.
12      print 'The square root of', number, 'is', square-root
13
14  # Call the main function.
15  main()
```

Program Output (with input shown in bold)

```
Enter a number: 25 [Enter]
The square root of 25 is 5.0
```

Program 6-10 shows another example that uses the `math` module. This program uses the `hypot` function to calculate the length of a right triangle's hypotenuse.

Program 6-10 (hypotenuse.py)

```
# This program calculates the length of a right
# triangle's hypotenuse.
import math

def main():
    # Get the length of the triangle's two sides.
    a = input('Enter the length of side A: ')
    b = input('Enter the length of side B: ')

    # Calculate the length of the hypotenuse.
    c = math.hypot(a, b)

    # Display the length of the hypotenuse.
    print 'The length of the hypotenuse is', c

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the length of side A: 5.0 [Enter]
Enter the length of side B: 12.0 [Enter]
The length of the hypotenuse is 13.0
```

Table 6-2 Many of the functions in the `math` module

math Module Function	Description
`acos(x)`	Returns the arc cosine of x, in radians.
`asin(x)`	Returns the arc sine of x, in radians.
`atan(x)`	Returns the arc tangent of x, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to x.
`cos(x)`	Returns the cosine of x in radians.
`degrees(x)`	Assuming x is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to x.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (x, y).
`log(x)`	Returns the natural logarithm of x.
`log10(x)`	Returns the base-10 logarithm of x.
`radians(x)`	Assuming x is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of x in radians.
`sqrt(x)`	Returns the square root of x.
`tan(x)`	Returns the tangent of x in radians.

The `math.pi` and `math.e` Values

The `math` module also defines two variables, `pi` and `e`, which are assigned mathematical values for pi and e. You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses `pi`. (Notice that we use dot notation to refer to the variable.)

```
area = math.pi * radius**2
```

Checkpoint

6.12 What `import` statement do you need to write in a program that uses the `math` module.

6.13 Write a statement that uses a `math` module function to get the square root of 100 and assigns it to a variable.

6.14 Write a statement that uses a `math` module function to convert 45 degrees to radians and assigns the value to a variable.

6.4 Storing Functions in Modules

CONCEPT: **A** module is a file that contains Python code. Large programs are easier to debug and maintain when they are divided into modules.

As your programs become larger and more complex, the need to organize your code becomes greater. You have already learned that a large and complex program should be divided into functions that each performs a specific task. As you write more and more functions in a program, you should consider organizing the functions by storing them in modules.

A module is simply a file that contains Python code. When you break a program into modules, each module should contain functions that perform related tasks. For example, suppose you are writing an accounting system. You would store all of the account receivable functions in their own module, all of the account payable functions in their own module, and all of the payroll functions in their own module. This approach, which is called *modularization*, makes the program easier to understand, test, and maintain.

Modules also make it easier to reuse the same code in more than one program. If you have written a set of functions that are needed in several different programs, you can place those functions in a module. Then, you can import the module in each program that needs to call one of the functions.

Let's look at a simple example. Suppose your instructor has asked you to write a program that calculates the following:

- The area of a circle
- The circumference of a circle
- The area of a rectangle
- The perimeter of a rectangle

There are obviously two categories of calculations required in this program: those related to circles, and those related to rectangles. You could write all of the circle-related functions in one module, and the rectangle-related functions in another module. Program 6-11 shows the `circle` module. The module contains two function definitions: `area` (which returns the area of a circle) and `circumference` (which returns the circumference of a circle).

Program 6-11 (circle.py)

```
# The circle module has functions that perform
# calculations related to circles.
```

```
import math

# The area function accepts a circle's radius as an
# argument and returns the area of the circle.
def area(radius):
    return math.pi * radius**2

# The circumference function accepts a circle's
# radius and returns the circle's circumference.
def circumference(radius):
    return 2 * math.pi * radius
```

Program 6-12 shows the `rectangle` module. The module contains two function definitions: `area` (which returns the area of a rectangle) and `perimeter` (which returns the perimeter of a rectangle.)

Program 6-12 (rectangle.py)

```
# The rectangle module has functions that perform
# calculations related to rectangles.

# The area function accepts a rectangle's width and
# length as arguments and returns the rectangle's area.
def area(width, length):
    return width * length

# The perimeter function accepts a rectangle's width
# and length as arguments and returns the rectangle's
# perimeter.
def perimeter(width, length):
    return 2 * (width + length)
```

Notice that both of these files contain function definitions, but they do not contain code that calls the functions. That will be done by the program or programs that import these modules.

Before continuing, we should mention the following things about module names:

- A module's file name should end in `.py`. If the module's file name does not end in `.py` you will not be able to import it into other programs.
- A module's name cannot be the same as a Python key word. An error would occur, for example, if you named a module `for`.

To use these modules in a program, you import them with the `import` statement. Here is an example of how we would import the `circle` module:

```
import circle
```

When the Python interpreter reads this statement it will look for the file `circle.py` in the same folder as the program that is trying to import it. If it finds the file it will load it into memory. If it does not find the file, an error occurs.[2]

Once a module is imported you can call its functions. Assuming that `radius` is a variable that is assigned the radius of a circle, here is an example of how we would call the `area` and `circumference` functions:

```
my_area = circle.area(radius)
my-circum = circle.circumference(radius)
```

Program 6-13 shows a complete program that uses these modules.

Program 6-13 (geometry.py)

```
# This program allows the user to choose various
# geometry calculations from a menu. This program
# imports the circle and rectangle modules.

import circle
import rectangle

# The main function.
def main():
    # The choice variable controls the loop
    # and holds the user's menu choice.
    choice = 0

    while choice != 5:
        # display the menu.
        display_menu()

        # Get the user's choice.
        choice = input('Enter your choice: ')

        # Perform the selected action.
        if choice == 1:
            radius = input("Enter the circle's radius: ")
            print 'The area is', circle.area(radius)
        elif choice == 2:
            radius = input("Enter the circle's radius: ")
            print 'The circumference is', \
                  circle.circumference(radius)
        elif choice == 3:
            width = input("Enter the rectangle's width: ")
            length = input("Enter the rectangle's length: ")
            print 'The area is', rectangle.area(width, length)
```

---

[2] Actually the Python interpreter is set up to look in various other predefined locations in your system when it does not find a module in the program's folder. If you choose to learn about the advanced features of Python, you can learn how to specify where the interpreter looks for modules.

```
        elif choice == 4:
            width = input("Enter the rectangle's width: ")
            length = input("Enter the rectangle's length: ")
            print 'The perimeter is', \
                  rectangle.perimeter(width, length)
        elif choice == 5:
            print 'Exiting the program...'
        else:
            print 'Error: invalid selection.'

# The display-menu function displays a menu.
def display-menu():
    print '       MENU'
   'print '1) Area of a circle'
    print '2) Circumference of a circle'
    print '3) Area of a rectangle'
    print '4) Perimeter of a rectangle'
    print '5) Quit'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
      MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 1 [Enter]
Enter the circle's radius: 10
The area is 314.159265359
      MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 2 [Enter]
Enter the circle's radius: 10
The circumference is 62.8318530718
      MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
```

(program output continues)

Program Output *(continued)*

```
Enter your choice: 3 [Enter]
Enter the rectangle's width: 5
Enter the rectangle's length: 10
The area is 50
       MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 4 [Enter]
Enter the rectangle's width: 5
Enter the rectangle's length: 10
The perimeter is 30
       MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 5 [Enter]
Exiting the program...
```

Menu Driven Programs

Program 6-13 is an example of a menu-driven program. A *menu-driven program* displays a list of the operations on the screen, and allows the user to select the operation that he or she wants the program to perform. The list of operations that is displayed on the screen is called a *menu*. When Program 6-13 is running, the user enters 1 to calculate the area of a circle, 2 to calculate the circumference of a circle, and so forth.

Once the user types a menu selection, the program uses a decision structure to determine which menu item the user selected. An `if-elif-else` statement is used in Program 6-13 (in lines 22 through 41) to carry out the user's desired action. The entire process of displaying a menu, getting the user's selection, and carrying out that selection is repeated by a `while` loop (which begins in line 14). The loop repeats until the user selects 5 (Quit) from the menu.

Review Questions

Multiple Choice

1. This is a prewritten function that is built into a programming language.
   a. standard function
   b. library function
   c. custom function
   d. cafeteria function

2. This term describes any mechanism that accepts input, performs some operation that cannot be seen on the input, and produces output.
   a. glass box
   b. white box
   c. opaque box
   d. black box
3. This standard library function returns a random integer within a specified range of values.
   a. `random`
   b. `randint`
   c. `random_integer`
   d. `uniform`
4. This standard library function returns a random floating-point number in the range of 0.0 up to 1.0 (but not including 1.0).
   a. `random`
   b. `randint`
   c. `random_integer`
   d. `uniform`
5. This standard library function returns a random floating-point number within a specified range of values.
   a. `random`
   b. `randint`
   c. `random_integer`
   d. `uniform`
6. This statement causes a function to end and sends a value back to the part of the program that called the function.
   a. `end`
   b. `send`
   c. `exit`
   d. `return`
7. This is a design tool that describes the input, processing, and output of a function.
   a. hierarchy chart
   b. IPO chart
   c. datagram chart
   d. data processing chart
8. This type of function returns either `True` or `False`.
   a. Binary
   b. `true_false`
   c. Boolean
   d. logical
9. This is `math` module function.
   a. `derivative`
   b. `factor`
   c. `sqrt`
   d. `differentiate`

10. A menu is a __________.
    a. case structure that selects an operation in a program
    b. group of modules that perform individual tasks
    c. list of operations displayed on the screen that the user may choose from
    d. table of Boolean choices

True or False

1. Some library functions are built into the Python interpreter.
2. You do not have to have an `import` statement in a program to use the functions in the `random` module.
3. Complex mathematical expressions can sometimes be simplified by breaking out part of the expression and putting it in a function.
4. A function in Python can return more than one value.
5. IPO charts provide only brief descriptions of a function's input, processing, and output, but do not show the specific steps taken in a function.

Short Answer

1. Suppose you want to select a random number from the following list:

   ```
   [0, 5, 10, 15, 20, 25, 30]
   ```

   What library function would you use?
2. What statement do you have to have in a value-returning function?
3. What three things are listed on an IPO chart?
4. What is a Boolean function?
5. What are the advantages of breaking a large program into modules?

Algorithm Workbench

1. Write a statement that generates a random number in the range of 1 through 100 and assigns it to a variable named `rand`.
2. The following statement calls a function named `half`, which returns a value that is half that of the argument. (Assume the `number` variable references a `float` value.) Write code for the function.

   ```
   result = half(number)
   ```
3. A program contains the following function definition:

   ```
   def cube(num):
   return num * num * num
   ```

   Write a statement that passes the value 4 to this function and assigns its return value to the variable `result`.
4. Write a function named `times_ten` that accepts a number as an argument. When the function is called, it should return the value of its argument multiplied times 10.
5. Write a function named `get_first_name` that asks the user to enter his or her first name, and returns it.

Programming Exercises

1. Feet to Inches

One foot equals 12 inches. Write a function named `feet_to_inches` that accepts a number of feet as an argument, and returns the number of inches in that many feet. Use the function in a program that prompts the user to enter a number of feet and then displays the number of inches in that many feet.

2. Math Quiz

Write a program that gives simple math quizzes. The program should display two random numbers that are to be added, such as:

```
  247
+ 129
```

The program should allow the student to enter the answer. If the answer is correct, a message of congratulations should be displayed. If the answer is incorrect, a message showing the correct answer should be displayed.

3. Maximum of Two Values

Write a function named `maximum` that accepts two integer values as arguments and returns the value that is the greater of the two. For example, if 7 and 12 are passed as arguments to the function, the function should return 12. Use the function in a program that prompts the user to enter two integer values. The program should display the value that is the greater of the two.

4. Falling Distance

The following formula can be used to determine the distance an object falls due to gravity in a specific time period, starting from rest:

$$d = \tfrac{1}{2} gt^2$$

The variables in the formula are as follows: d is the distance in meters, g is 9.8, and t is the amount of time in seconds, that the object has been falling.

Write a function named `falling_distance` that accepts an object's falling time in seconds as an argument. The function should return the distance in meters that the object has fallen during that time interval. Write a program that calls the function in a loop that passes the values 1 through 10 as arguments and displays the return value.

5. Kinetic Energy

In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

$$KE = \tfrac{1}{2} mv^2$$

The variables in the formula are as follows: KE is the kinetic energy in joules, m is the object's mass in kilograms, and v is the object's velocity in meters per second.

Write a function named `kinetic_energy` that accepts an object's mass in kilograms and velocity in meters per second as arguments. The function should return the amount of kinetic energy that the object has. Write a program that asks the user to enter values for mass and velocity, and then calls the `kinetic_energy` function to get the object's kinetic energy.

6. Test Average and Grade

Write a program that asks the user to enter five test scores. The program should display a letter grade for each score and the average test score. Write the following functions in the program:

- `calc_average`— This function should accept five test scores as arguments and return the average of the scores.
- `determine_grade`— This function should accept a test score as an argument and return a letter grade for the score, based on the following grading scale:

Score	Letter Grade
`90–100`	`A`
`80–89`	`B`
`70–79`	`C`
`60–69`	`D`
`Below 60`	`F`

7. Odd/Even Counter

In this chapter you saw an example of how to write an algorithm that determines whether a number is even or odd. Write a program that generates 100 random numbers, and keeps a count of how many of those random numbers are even and how many are odd.

8. Prime Numbers

A prime number is a number that is only evenly divisible by itself and 1. For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a Boolean function named `is_prime` which takes an integer as an argument and returns `True` if the argument is a prime number, or `False` otherwise. Use the function in a program that prompts the user to enter a number and then displays a message indicating whether the number is prime.

TIP: Recall that the `%` operator divides one number by another and returns the remainder of the division. In an expression such as `num1 % num2`, the `%` operator will return 0 if `num1` is evenly divisible by `num2`.

9. Prime Number List

This exercise assumes you have already written the `is_prime` function in Programming Exercise 8. Write another program that displays all of the prime numbers from 1 through 100. The program should have a loop that calls the `is_prime` function.

10. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows.

1. When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the com-

puter has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)

2. The user enters his or her choice of "rock", "paper", or "scissors" at the keyboard.
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
   - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
   - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
   - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
   - If both players make the same choice, the game must be played again to determine the winner.

CHAPTER

7 Files and Exceptions

TOPICS

7.1 Introduction to File Input and Output
7.2 Using Loops to Process Files
7.3 Processing Records
7.4 Exceptions

7.1 Introduction to File Input and Output

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs, because data that is stored in RAM (referenced by variables) disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be retrieved and used at a later time.

Most of the commercial software packages that you use on a day-to-day basis store data in files. The following are a few examples.

- Word processors. Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- Image editors. Image editing programs are used to draw graphics and edit images such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- Spreadsheets. Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved in a file for use later.
- Games. Many computer games keep data stored in files. For example, some games keep a list of player names with their scores stored in a file. These games typically

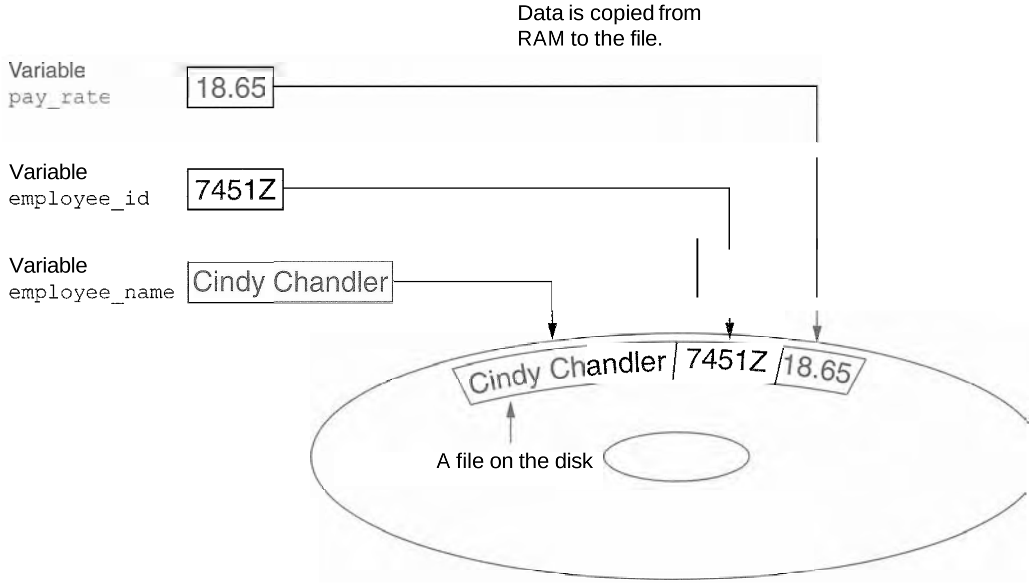
display the players' names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game and then resume playing it later without having to start from the beginning.

- Web browers. Sometimes when you visit a Web page, the browser stores a small file known as a cookie on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, accounting systems keep data about a company's financial operations in files, and so on.

Programmers usually refer to the process of saving data in a file as "writing data to" the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 7-1. The term *output file* is used to describe a file that data is written to. It is called an output file because the program stores output in it.

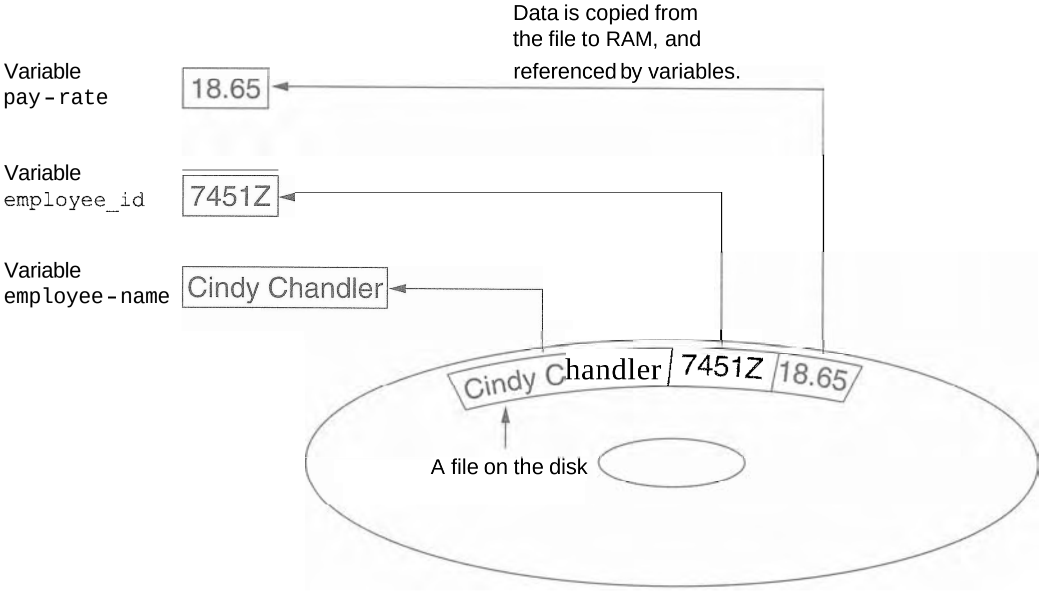
Figure 7-1 Writing data to a file



Data is copied from RAM to the file.
Variable
pay_rate
18.65
Variable
employee_id
7451Z
Variable
employee_name
Cindy Chandler
Cindy Chandler
7451Z
18.65
A file on the disk

The process of retrieving data from a file is known as "reading data from" the file. When a piece of data is read from a file, it is copied from the file into RAM, and referenced by a variable. Figure 7-2 illustrates this. The term *input file* is used to describe a file that data is read from. It is called an input file because the program gets input from the file.

Figure 7-2 Reading data from a file


Data is copied from
the file to RAM, and
referenced by variables.
Variable
pay-rate
18.65
Variable
employee_id
7451Z
Variable
employee-name
Cindy Chandler
Cindy Chandler
7451Z
18.65
A file on the disk

This chapter discusses how to write data to files and read data from files. There are always three steps that must be taken when a file is used by a program.

1. Open the file—Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. Process the file—In this step data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. Close the file—When the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

Types of Files

In general, there are two types of files: text and binary. A text file contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A binay file contains data that has not been converted to text. As a consequence, you cannot view the contents of a binary file with a text editor.

Although Python allows you to work both text files and binary files, we will work only with text files in this book. That way, you will be able to use an editor such as Notepad to inspect the files that your programs create.

File Access Methods

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access. When you work with a sequential access file, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a direct access file (which is also known as a random access file), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to listen to.

In this book we will use sequential access files, Sequential access files are easy to work with, and you can use them to gain an understanding of basic file operations.

Filenames and File Objects

Most computer users are accustomed to the fact that files are identified by a filename. For example, when you create a document with a word processor and then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 7-3 shows how three files named `cat.jpg`, `notes.txt`, and `resume.doc` might be represented in Windows Explorer.

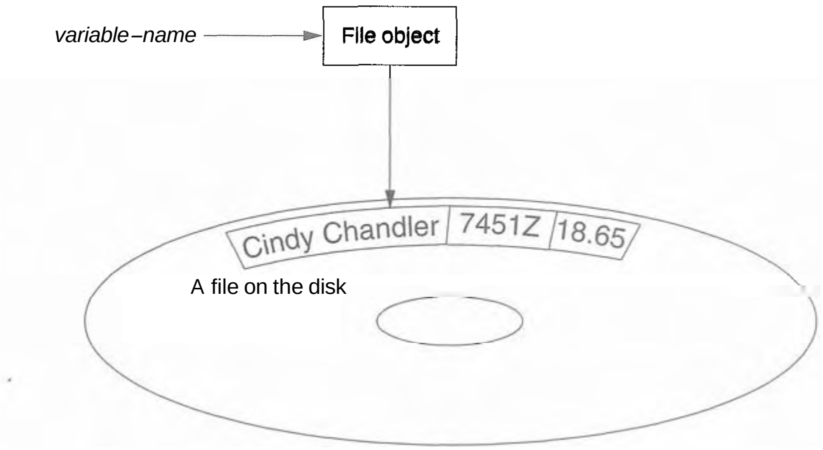
Figure 7-3 Three files

cat.jpg
notes.txt
resume.doc

Each operating system has its own rules for naming files. Many systems support the use of filename extensions, which are short sequences of characters that appear at the end of a filename preceded by a period (which is known as a "dot"). For example, the files depicted in Figure 7–3 have the extensions `.jpg`, `.txt`, and. `doc`. The extension usually indicates the type of data stored in the file. For example, the `.jpg` extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The `.txt` extension usually indicates that the file contains text. The `.doc` extension usually indicates that the file contains a Microsoft Word document.

In order for a program to work with a file on the computer's disk, the program must create a file object in memory. A file object is an object that is associated with a specific file, and provides a way for the program to work with that file. In the program, a variable references the file object. This variable is used to carry out any operations that are performed on the file. This concept is shown in Figure 7-4.

Figure 7-4 A variable name references a file object that is associated with a file



Opening a File

You use the `open` function in Python to open a file. The `open` function creates a file object and associates it with a file on the disk. Here is the general format of how the `open` function is used:

```
file-variable = open(filename, mode)
```

In the general format:

- *file_variable* is the name of the variable that will reference the file object.
- *filename* is a string specifying the name of the file.
- *mode* is a string specifying the mode (reading, writing, etc.) in which the file will be opened. Table 7-1 shows three of the strings that you can use to specify a mode. (There are other, more complex modes. The modes shown in Table 7-1 are the ones we will use in this book.)

Table 7-1 Some of the Python file modes

Mode	**Description**
`'r'`	Open a file for reading only. The file cannot be changed or written to.
`'w'`	Open a file for writing. If the file already exists, erase its contents. If it does not exist, create it.
`'a'`	Open a file to be written to. All data written to the file will be appended to its end. If the file does not exist, create it.

For example, suppose the file `customers.txt` contains customer data, and we want to open for reading. Here is an example of how we would call the `open` function:

```
customer-file = open('cusomters.txt', 'r')
```

After this statement executes, the file named `customers.txt` will be opened, and the variable `customer_file` will reference a file object that we can use to read data from the file.

Suppose we want to create a file named `sales.txt` and write data to it. Here is an example of how we would call the `open` function:

```
sales-file = open('sales.txt', 'w')
```

After this statement executes, the file named `sales.txt` will be created, and the variable `sales_file` will reference a file object that we can use to write data to the file.

> **WARNING:** Remember, when you use the `'w'` mode you are creating the file on the disk. If a file with the specified name already exists when the file is opened, the contents of the existing file will be erased.

Writing Data to a File

So far in this book you have worked with several of Python's library functions, and you have even written your own functions. Now we will introduce you to another type of function, which is known as a method. A method is a function that belongs to an object, and performs some operation using that object. Once you have opened a file, you use the file object's methods to perform operations on the file.

For example, file objects have a method named `write` that can be used to write data to a file. Here is the general format of how you call the `write` method:

```
file_variable.write(string)
```

In the format, *`file_variable`* is a variable that references a file object, and *`string`* is a string that will be written to the file. The file must be opened for writing (using the `'w'` or `'a'` mode) or an error will occur.

Let's assume that `customer_file` references a file object, and the file was opened for writing with the `'w'` mode. Here is an example of how we would write the string 'Charles Pace' to the file:

```
customer_file.write('Charles Pace')
```

The following code shows another example:

```
name = 'Charles Pace'
customer_file.write(name)
```

The second statement writes the value referenced by the name variable to the file associated with `customer_file`. In this case, it would write the string 'Charles Pace' to the file. (These examples show a string being written to a file, but you can also write numeric values.)

Once a program is finished working with a file, it should close the file. Closing a file disconnects the program from the file. In some systems, failure to close an output file can cause a loss of data. This happens because the data that is written to a file is first written to a buffer, which is a small "holding section" in memory. When the buffer is full, the system writes the buffer's contents to the file. This technique increases the system's performance, because writing data to memory is faster than writing it to a disk. The process of closing an output file forces any unsaved data that remains in the buffer to be written to the file.

In Python you use the file object's `close` method to close a file. For example, the following statement closes the file that is associated with `customer_file`:

```
customer_file.close()
```

Program 7-1 shows a complete Python program that opens an output file, writes data to it, and then closes it.

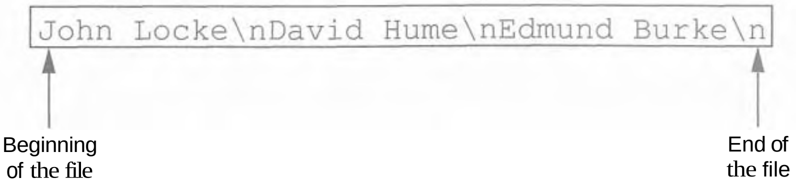
Program 7-1 (file_write.py)

```
 1  # This program writes three lines of data
 2  # to a file.
 3  def main():
 4      # Open a file named philosophers.txt.
 5      outfile = open('philosophers.txt', 'w')
 6
 7      # Write the names of three philosphers
 8      # to the file.
 9      outfile.write('John Locke\n')
10      outfile.write('David Hume\n')
11      outfile.write('Edmund Burke\n')
12
13      # Close the file.
14      outfile.close()
15
16  # Call the main function.
17  main()
```

Line 5 opens the file `philosophers.txt` using the `'w'` mode. (This causes the file to be created, and opens it for writing.) It also creates a file object in memory and assigns that object to the `outfile` variable.

The statements in lines 9 through 11 write three strings to the file. Line 9 writes the string `'John Locke\n'`, line 10 writes the string `'David Hume\n'`, and line 11 writes the string `'Edmund Burke\n'`. Line 14 closes the file. After this program runs, the three items shown in Figure 7-5 will be written to the `philosophers.txt` file.

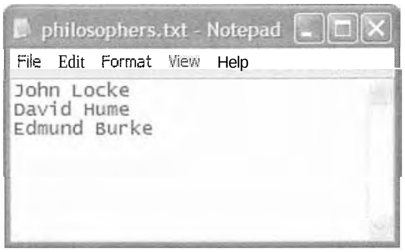
Figure 7-5 Contents of the file `philosophers.txt`



```
John Locke\nDavid Hume\nEdmund Burke\n
```

Notice that each of the strings written to the file end with `\n`, which you will recall is the newline escape sequence. The `\n` not only separates the items that are in the file, but also causes each of them to appear in a separate line when viewed in a text editor. For example, Figure 7-6 shows the `philosophers.txt` file as it appears in Notepad.

Figure 7-6 Contents of `philosophers.txt` in Notepad



```
John Locke
David Hume
Edmund Burke
```

Reading Data From a File

If a file has been opened for reading (using the `'r'` mode) you can use the file object's `read` method to read its entire contents into memory. When you call the `read` method, it returns the file's contents as a string. For example, Program 7-2 shows how we can use the `read` method to read the contents of the `philosophers.txt` file that we created earlier.

Program 7-2 (file_read.py)

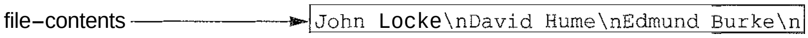
```
# This program reads and displays the contents
# of the philosophers.txt file.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read the file's contents.
    file-contents = infile.read()

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print file-contents

# Call the main function.
main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

The statement in line 5 opens the `philosophers.txt` file for reading, using the `'r'` mode. It also creates a file object and assigns the object to the `infile` variable. Line 8 calls the `infile.read` method to read the file's contents. The file's contents are read into memory as a string and assigned to the `file_contents` variable. This is shown in Figure 7-7. Then the statement in line 15 prints the string that is referenced by the variable.

Figure 7-7 The `file_contents` variable references the string that was read from the file



Although the `read` method allows you to easily read the entire contents of a file with one statement, many programs need to read and process the items that are stored in a file one at a time. For example, suppose a file contains a series of sales amounts, and you need to write a program that calculates the total of the amounts in the file. The program would read each sale amount from the file and add it to an accumulator.

In Python you can use the `readline` method to read a line from a file. (A line is simply a string of characters that are terminated with a `\n`.) The method returns the line as a string, including the `\n`. Program 7-3 shows how we can use the `readline` method to read the contents of the `philosophers.txt` file, one Line at a time.

Program 7-3 (line_read.py)

```
# This program reads the contents of the
# philosophers.txt file one line at a time.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read three lines from the file.
    line1 = infile.readline()
    line2 = infile.readline()
    line3 = infile.readline()

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print line1
    print line2
    print line3

# Call the main function.
main()
```

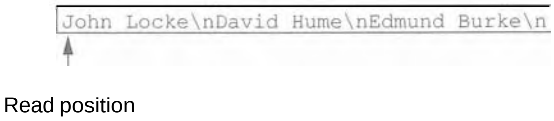
Program Output

```
John Locke

David Hume

Edmund Burke
```

Before we examine the code, notice that a blank line is displayed after each line in the output. This is because each item that is read from the file ends with a newline character (`\n`). Later you will learn how to remove the newline character.

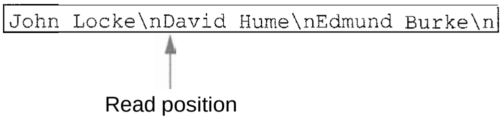
The statement in line 5 opens the `philosophers.txt` file for reading, using the `'r'` mode. It also creates a file object and assigns the object to the `infile` variable. When a file is opened for reading, a special value known as a read ***position*** is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. Initially, the read position is set to the beginning of the file. After the statement in line 5 executes, the read position for the `philosophers.txt` file will be positioned as shown in Figure 7-8.

Figure 7-8 Initial read position



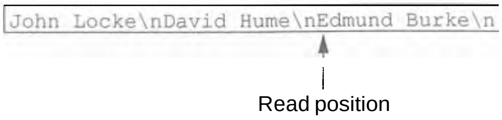
The statement in line 8 calls the `infile.readline` method to read the first line from the file. The line, which is returned as a string, is assigned to the `line1` variable. After this statement executes the `line1` variable will be assigned the string `'John Locke\n'`. In addition, the file's read position will be advanced to the next line in the file, as shown in Figure 7-9.

Figure 7-9 Read position *advanced* to *the next* line



Then the statement in line 9 reads the next line from the file and assigns it to the `line2` variable. After this statement executes the `line2` variable will reference the string `'David Hume\n'`. The file's read position will be advanced to the next line in the file, as shown in Figure 7-10.

Figure 7-10 Read position *advanced* to *the* next line



Then the statement in line 10 reads the next line from the file and assigns it to the `line3` variable. After this statement executes the `line3` variable will reference the string `'Edmund Burke\n'`. After this statement executes, the read position will be advanced to the end of the file, as shown in Figure 7-11. Figure 7-12 shows the `line1`, `line2`, and `line3` variables and the strings they reference after these statements have executed.

Figure 7-11 Read position advanced to the end of the file

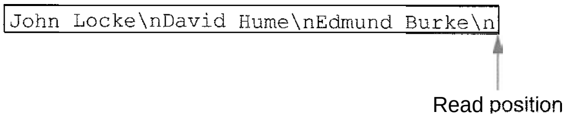
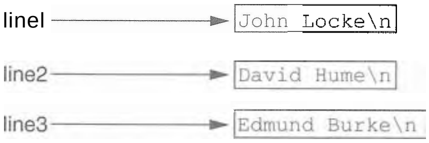


Figure 7-82 The strings referenced by the `line1`, `line2`, and `line3` variables



The statement in line 13 closes the file. The `print` statements in lines 17 through 19 display the contents of the `line1`, `line2`, and `line3` variables.

> **NOTE:** If the last line in a file is not terminated with a `\n`, the `readline` method will return the line without a `\n`.

Concatenating a Newline to a String

Program 7–1 wrote three string literals to a file, and each string literal ended with a `\n` escape sequence. In most cases, the data items that are written to a file are not string literals, but values in memory that are referenced by variables. This would be the case in a program that prompts the user to enter data, and then writes that data to a file.

When a program writes data that has been entered by the user to a file, it is usually necessary to concatenate a `\n` escape sequence to the data before writing it. This ensures that each piece of data is written to a separate line in the file. Program 7-4 demonstrates how this is done.

Program 7-4 (write_names.py)

```
# This program gets three names from the user
# and writes them to a file.

def main():
    # Get three names.
    print 'Enter the names of three friends.'
    name1 = raw-input('Friend #1: ')
```

(program continues)

Program 7-4 *(continued)*

```
 8      name2 = raw_input('Friend #2: ')
 9      name3 = raw_input('Friend #3: ')
10
11      # Open a file named friends.txt.
12      myfile = open('friends.txt', 'w')
13
14      # Write the names to the file.
15      myfile.write(name1 + '\n')
16      myfile.write(name2 + '\n')
17      myfile.write(name3 + '\n')
18
19      # Close the file.
20      myfile.close()
21      print 'The names were written to friends.txt.'
22
23  # Call the main function.
24  main()
```

Program Output (with input shown in bold)

```
Enter the names of three friends.
Friend #1: Joe [Enter]
Friend #2: Rose [Enter]
Friend #3: Geri [Enter]
The names were written to friends.txt.
```

Lines 7 though 9 prompt the user to enter three names, and those names are assigned to the variables `name1`, `name2`, and `name3`. Line 12 opens a file named `friends.txt` for writing. Then, lines 15 through 17 write the names entered by the user, each with `'\n'` concatenated to it. As a result, each name will have the `\n` escape sequence added to it when written to the file. Figure 7-13 shows the contents of the file with the names entered by the user in the sample run.

Figure 7-13 The friends.txt file

```
Joe\nRose\nGeri\n
```

Reading a String and Stripping the Newline From It

Sometimes complications are caused by the `\n` that appears at the end of the strings that are returned from the `readline` method. For example, did you notice in the sample output of Program 7-3 that a blank line is printed after each line of output? This is because each of the strings that are printed in lines 17 through 19 end with a `\n` escape sequence. When the strings are printed, the `\n` causes an extra blank line to appear.

The `\n` serves a necessary purpose inside a file: it separates the items that are stored in the file. However, in many cases you want to remove the `\n` from a string after it is read from a file. Each string in Python has a method named `rstrip` that removes, or "strips," specific characters from the end of a string. (It is named `rstrip` because it strips characters from the right side of a string.) The following code shows an example of how the `rstrip` method can be used.

```
name = 'Joanne Manchester\n'
name = name.rstrip('\n')
```

The first statement assigns the string `'Joanne Manchester\n'` to the name variable. (Notice that the string ends with the `\n` escape sequence.) The second statement calls the `name.rstrip('\n')` method. The method returns a copy of the name string without the trailing `\n`. This string is assigned back to the name variable. The result is that the trailing `\n` is stripped away from the name string.

Program 7-5 is another program that reads and displays the contents of the `philosophers.txt` file. This program uses the `rstrip` method to strip the `\n` from the strings that are read from the file before they are displayed on the screen. As a result, the extra blank lines do not appear in the output.

Program 7-5 (strip_newline.py)

```
# This program reads the contents of the
# philosophers.txt file one line at a time.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read three lines from the file.
    linel = infile.readline()
    line2 = infile.readline()
    line3 = infile.readline()

    # Strip the \n from each string.
    linel = linel.rstrip('\n')
    Pine2 = line2.rstrip('\n')
    line3 = line3.rstrip('\n')

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print linel
    print line2
    print line3

# Call the main function.
main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

Appending Data to an Existing File

When you use the `'w'` mode to open an output file and a file with the specified filename already exists on the disk, the existing file will be erased and a new empty file with the same name will be created. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.

In Python you can use the `'a'` mode to open an output file in append mode, which means the following.

- If the file already exists, it will not be erased. If the file does not exist, it will be created.
- When data is written to the file, it will be written at the end of the file's current contents.

For example, assume the file `friends.txt` contains the following names, each in a separate line:

```
Joe
Rose
Geri
```

The following code opens the file and appends additional data to its existing contents.

```
myfile = open('friends.txt', 'a')
myfile.write('Matt\n')
myfile.write('Chris\n')
myfile.write('Suze\n')
myfile.close()
```

After this program runs, the file `friends.txt` will contain the following data:

```
Joe
Rose
Geri
Matt
Chris
Suze
```

Writing and Reading Numeric Data

Strings can be written directly to a file with the `write` method, but numbers must be converted to strings before they can be written. Python has a built-in function named `str` that converts a value to a string. For example, assuming the variable num is assigned the value 99, the expression `str(num)` will return the string `'99'`.

Program 7-6 shows an example of how you can use the `str` function to convert a number to a string, and write the resulting string to a file.

Program 7-6 (write_numbers.py)

```
 1  # This program demonstrates how numbers
 2  # must be converted to strings before they
 3  # are written to a text file.
 4
 5  def main():
 6      # Open a file for writing.
 7      outfile = open('numbers.txt', 'w')
 8
 9      # Get three numbers from the user.
10      numl = input('Enter a number: ')
11      num2 = input('Enter another number: ')
12      num3 = input('Enter another number: ')
13
14      # Write the numbers to the file.
15      outfile.write(str(num1) + '\n')
16      outfile.write(str(num2) + '\n')
17      outfile.write(str(num3) + '\n')
18
19      # Close the file.
20      outfile.close()
21      print 'Data written to numbers.txt'
22
23  # Call the main function.
24  main()
```

Proaram Outpout (with **input** shown in bold)

```
Ent                22 [Enter]
Enter anotner number: 14 [Enter]
Enter another nun           99 [Enter]
Data written to numbers.txt
```

The statement in line 7 opens the file `numbers.txt` for writing. Then the statements in lines 10 through 12 prompt the user to enter three numbers, which are assigned to the variables `num1`, `num2`, and `num3`.

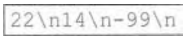
Take a closer look at the statement in line 15, which writes the value referenced by `num1` to the file:

```
outfile.write(str(numl) + '\n')
```

The expression `str(numl) + '\n'` converts the value referenced by `num1` to a string and concatenates the `\n` escape sequence to the string. In the program's sample run, the user entered 22 as the first number, so this expression produces the string `'22\n'`. As a result, the string `'22\n'` is written to the file.

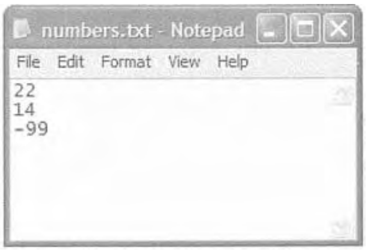
Lines 16 and 17 perform the similar operations, writing the values referenced by `num2` and `num3` to the file. After these statements execute, the values shown in Figure 7-14 will be written to the file. Figure 7-15 shows the file viewed in Notepad.

Figure 7-14 Contents of the `numbers.txt` file



```
22\n14\n-99\n
```

Figure 7-15 The `numbers.txt` file viewed in Notepad



```
22
14
-99
```

When you read numbers from a text file, they are always read as strings. For example, suppose a program uses the following code to read the first line from the `numbers.txt` file that was created by Program 7-6:

```
1 infile = open('numbers.txt', 'r')
2 value = infile.readline()
3 infile.close()
```

The statement in line 2 uses the `readline` method to read a line from the file. After this statement executes, the `value` variable will reference the string `'22\n'`. This can cause a problem if we intend to perform math with the `value` variable, because you cannot perform math on strings. In such a case you must convert the string to a numeric type.

Recall from Chapter 2 that Python provides the built-in function `int` to convert a string to an integer, and the built-in function `float` to convert a string to a floating-point number. For example, we could modify the code previously shown as follows:

```
1 infile = open('numbers.txt', 'r')
2 string_input = infile.readline()
3 value = int(string_input)
4 infile.close()
```

The statement in line 2 reads a line from the file and assigns it to the `string_input` variable. As a result, `string_input` will reference the string `'22\n'`. Then the statement in line **3** uses the `int` function to convert `string_input` to an integer, and assigns the result to `value`. After this statement executes, the `value` variable will reference the integer 22. (Both the `int` and `float` functions ignore any `\n` at the end of the string that is passed as an argument.)

This code demonstrates the steps involved in reading a string from a file with the `readline` method, and then converting that string to an integer with the `int` function. In many situations, however, the code can be simplified. A better way is to read the string from the file and convert it in one statement, as shown here:

```
1 infile = open('numbers.txt', 'r')
2 value = int(infile.readline())
3 infile.close()
```

Notice in line 2 that a call to the `readline` method is used as the argument to the `int` function. Here's how the code works: the `readline` method is called, and it returns a string. That string is passed to the `int` function, which converts it to an integer. The result is assigned to the `value` variable.

Program 7-7 shows a more complete demonstration. The contents of the `numbers.txt` file are read, converted to integers, and added together.

Program 7-7 (read_numbers.py)

```
# This program demonstrates how numbers that are
# read from a file must be converted from strings
# before they are used in a math operation.

def main():
    # Open a file for reading.
    infile = open('numbers.txt', 'r')

    # Read three numbers from the file.
    num1 = int(infile.readline())
    num2 = int(infile.readline())
    num3 = int(infile.readline())

    # Close the file.
    infile.close()

    # Add the three numbers.
    total = num1 + num2 + num3

    # Display the numbers and their total.
    print 'The numbers are:', num1, num2, num3
    print 'Their total is:', total

# Call the main function.
main()
```

Program Output

```
The numbers are: 22 14 -99
Their total is: -63
```

Checkpoint

7.1 What is an output file?

7.2 What is an input file?

7.3 What three steps must be taken by a program when it uses a file?

7.4 In general, what are the two types of files? What is the difference between these two types of files?

7.5 What are the two types of file access? What is the difference between these two?

7.6 When writing a program that performs an operation on a file, what two file-associated names do you have to work with in your code?

7.7 If a file already exists what happens to it if you try to open it as an output file (using the `'w'` mode)?

7.8 What is the purpose of opening a file?

7.9 What is the purpose of closing a file?

7.10 What is a file's read position? Initially, where is the read position when an input file is opened?

7.11 In what mode do you open a file if you want to write data to it, but you do not want to erase the file's existing contents? When you write data to such a file, to what part of the file is the data written?

7.2 Using Loops to Process Files

CONCEPT: Files usually hold large amounts of data, and programs typically use a loop to process the data in a file.

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in Program 7-8. This program gets sales amounts for a series of days from the user and writes those amounts to a file named `sales.txt`. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. Figure 7-16 shows the contents of the `sales.txt` file containing the data entered by the user in the sample run.

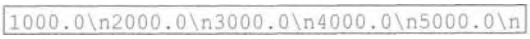
Program 7-8 (write_sales.py)

```
# This program prompts the user for sales amounts
# and writes those amounts to the sales.txt file.

def main():
    # Get the number of days.
    num_days = input('For how many days do ' + \
                     'you have sales? ')

    # Open a new file named sales.txt.
    sales-file = open('sales.txt', 'w')

    # Get the amount of sales for each day and write
    # it to the file.
    for count in range(1, num_days + 1):
        # Get the sales for a day.
        sales = input('Enter the sales for day #' + \
                      str(count) + ': ')

        # Write the sales amount to the file.
        sales_file.write(str(sales) + '\n')

    # Close the file.
    sales_file.close()
    print 'Data written to sales.txt.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
For how many days do you have sales? 5 [Enter]
Enter the sales for day #1: 1000.0 [Enter]
Enter the sales for day #2: 2000.0 [Enter]
Enter the sales for day #3: 3000.0 [Enter]
Enter the sales for day #4: 4000.0 [Enter]
Enter the sales for day #5: 5000.0 [Enter]
Data written to sales.txt.
```

Figure 7-16 Contents of the `sales.txt` file



```
1000.0\n2000.0\n3000.0\n4000.0\n5000.0\n
```

Reading a File with a Loop and Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, the `sales.txt` file that was created by Program 7-8 can have any number of items stored in it, because the program asks the user for the number of days that he or she has sales amounts for. If the user enters 5 as the number of days, the program gets 5 sales amounts and writes them to the file. If the user enters 100 as the number of days, the program gets 100 sales amounts and writes them to the file.

This presents a problem if you want to write a program that processes all of the items in the file, however many there are. For example, suppose you need to write a program that reads all of the amounts in the `sales.txt` file and calculates their total. You can use a loop to read the items in the file, but you need a way of knowing when the end of the file has been reached.

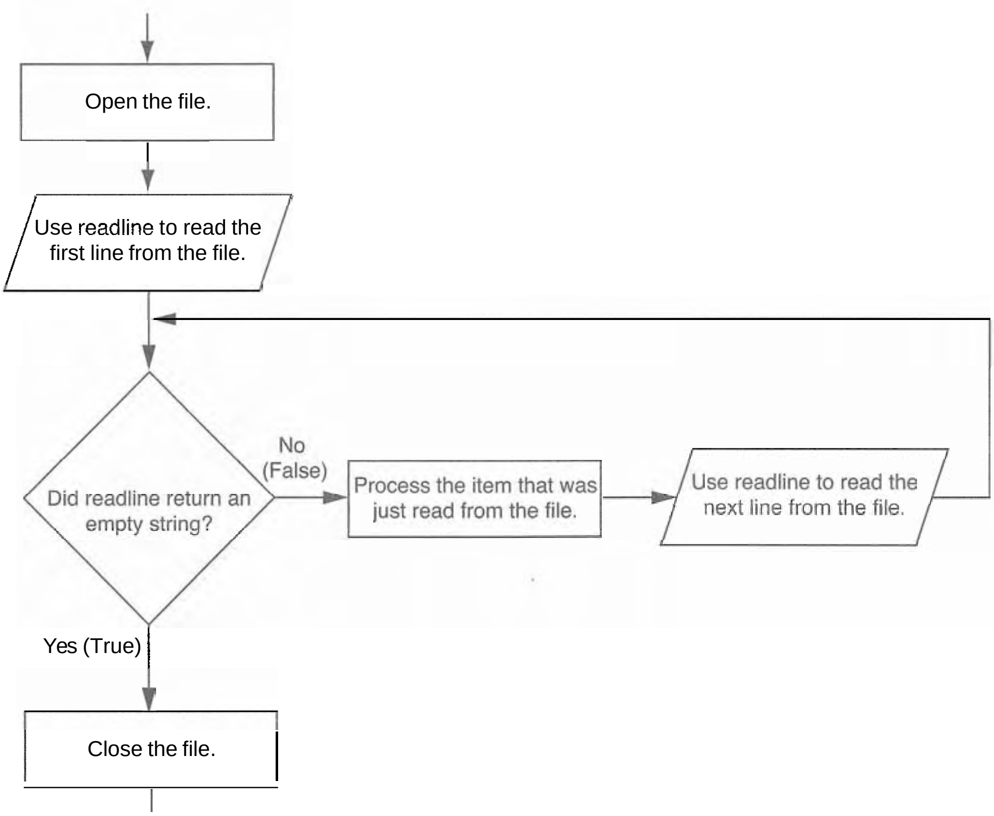
In Python, the `readline` method returns an empty string `('')` when it has attempted to read beyond the end of a file. This makes it possible to write a `while` loop that determines when the end of a file has been reached. Here is the general algorithm, in pseudocode:

```
Open the file
Use readline to read the first line from the file
While the value returned from readline is not an empty string:
    Process the item that was just read from the file
    Use readline to read the next line from the file.
Close the file
```

NOTE: In this algorithm we call the `readline` method just before entering the `while` loop. The purpose of this method call is to get the first line in the file, so it can be tested by the loop. This initial read operation is called a *priming read.*

Figure 7-17 shows this algorithm in a flowchart.

Figure 7-17 General logic for detecting the end of a file


Open the file.
Use readline to read the first line from the file.
Did readline return an empty string?
No (False)
Process the item that was just read from the file.
Use readline to read the next line from the file.
Yes (True)
Close the file.

Program 7-9 demonstrates how this can be done in code. The program reads and displays all of the values in the `sales.txt` file.

Program 7-9 (read_sales.py)

```
# This program reads all of the values in
# the sales.txt file.

def main():
    # Open the sales.txt file for reading.
    sales-file = open('sales.txt', 'r')

    # Read the first line from the file, but
    # don't convert to a number yet. We still
    # need to test for an empty string.
    line = sales_file.readline()

    # As long as an empty string is not returned
    # from readline, continue processing.
    while line != '':
        # Convert line to a float.
        amount = float(line)

        # Format and display the amount.
        print '$%.2f' % amount

        # Read the next line.
        line = sales-file.readline()

    # Close the file.
    sales_file.close()

# Call the main function.
main()
```

Program Output

```
$1000.00
$2000.00
$3000.00
$4000.00
$5000.00
```

Using Python's `for` Loop to Read Lines

In the previous example you saw how the `readline` method returns an empty string when the end of the file has been reached. Most programming languages provide a similar technique for detecting the end of a file. If you plan to learn programming languages other than Python, it is important for you to know how to construct this type of logic.

The Python language also allows you to write a `for` loop that automatically reads line in a file without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. When you simply want to read the lines in a file, one after the other, this technique is simpler and more elegant than writing a `while` loop that explicitly tests for an end of the file condition. Here is the general format of the loop:

```
for variable in file-object:
    statement
    statement
    etc.
```

In the general format, *variable* is the name of a variable and *file—object* is a variable that references a file object. The loop will iterate once for each line in the file. The first time the loop iterates, *variable* will reference the first line in the file (as a string), the second time the loop iterates, *variable* will reference the second line, and so forth. Program 7-10 provides a demonstration. It reads and displays all of the items in the `sales.txt` file.

Program 7-10 (read_sales2.py)

```
# This program uses the for loop to read
# all of the values in the sales·txt file.

def main():
    # Open the sales.txt file for reading.
    sales-file = open('sales.txt', 'r')

    # Read all the lines from the file.
    for line in sales-file:
        # Convert line to a float.
        amount = float(line)
        # Format and display the amount.
        print '$%.2f' % amount

    # Close the file.
    sales_file.close()

# Call the main function.
main()
```

Program Output

```
$1000.00
$2000.00
$3000.00
$4000.00
$5000.00
```

In the Spotlight:

Working with Files

Kevin is a freelance video producer who makes TV commercials for local businesses. When he makes a commercial, he usually films several short videos. Later, he puts these short videos together to make the final commercial. He has asked you to write the following two programs.

1. A program that allows him to enter the running time (in seconds) of each short video in a project. The running times are saved to a file.
2. A program that reads the contents of the file, displays the running times, and then displays the total running time of all the segments.

Here is the general algorithm for the first program, in pseudocode:

```
Get the number of videos in the project.
Open an output file.
For each video in the project:
    Get the video's running time.
    Write the running time to the file.
Close the file.
```

Program 7-11 shows the code for the first program.

Program 7-11 (save_running_times.py)

```
# This program saves a sequence of video running times
# to the video_times.txt file.

def main():
    # Get the number of videos in the project.
    num_videos = input('How many videos are in the project? ')

    # Open the file to hold the running times.
    video_file = open('video_times.txt', 'w')

    # Get each video's running time and write
    # it to the file.
    print 'Enter the running times for each video.'
    for count in range(1, num_videos + 1):
        run_time = input('Video #' + str(count) + ': ')
        video_file.write(str(run_time) + '\n')

    # Close the file.
    video_file.close()
    print 'The times have been saved to video_times.txt.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
How many videos are in the project? 6 [Enter]
Enter the running times for each video.
Video #1: 24.5 [Enter]
Video #2: 12.2 [Enter]
Video #3: 14.6 [Enter]
Video #4: 20.4 [Enter]
Video #5: 22.5 [Enter]
Video #6: 19.3 [Enter]
The times have been saved to video_times.txt.
```

Here is the general algorithm for the second program:

```
Initialize an accumulator to 0.
Initialize a count variable to 0.
Open the input file.
For each line in the file:
    Convert the line to a floating-point number. (This is the running time for a video.)
    Add one to the count variable. (This keeps count of the number of videos.)
    Display the running time for this video.
    Add the running time to the accumulator.
Close the file.
Display the contents of the accumulator as the total running time.
```

Program 7-12 shows the code for the second program.

Program 7-12 (read_running_times.py)

```
# This program the values in the video_times.txt
# file and calculates their total.

def main():
    # Open the video_times.txt file for reading.
    video_file = open('video_times.txt', 'r')

    # Initialize an accumulator to 0.0.
    total = 0.0

    # Initialize a variable to keep count of the videos.
    count = 0

    print 'Here are the running times for each video:'

    # Get the values from the file and total them.
    for line in video_file:
        # Convert a line to a float.
        run_time = float(line)

        # Add 1 to the count variable.
        count += 1

        # Display the time.
        print 'Video #' + str(count) + ': ' + str(run_time)

        # Add the time to total.
        total += run_time

    # Close the file.
    video_file.close()

    # Display the total of the running times.
    print 'The total running time is', total, 'seconds.'

# Call the main function.
main()
```

Program Output

```
Here are the running times for each video:
Video #1: 24.5
Video #2: 12.2
Video #3: 14.6
Video #4: 20.4
Video #5: 22.5
Video #6: 19.3
The total running time is 113.5 seconds.
```

Checkpoint

7.12 Write a short program that uses a `for` loop to write the numbers 1 through 10 to a file.

7.13 What does it mean when the `readline` method returns an empty string?

7.14 Assume that the file `data.txt` exists and contains several lines of text. Write a short program using the `while` loop that displays each line in the file.

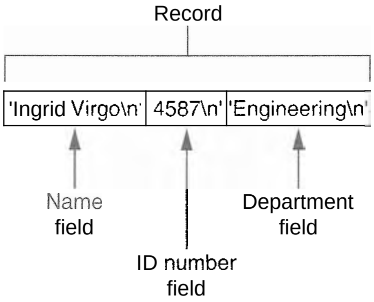
7.15 Revise the program that you wrote for Checkpoint 7.14 so to use the `for` loop instead of the `while` loop.

7.3 Processing Records

CONCEPT: The data that is stored in a file is frequently organized in records. A record is a complete set of data about an item, and a field is an individual piece of data within a record.

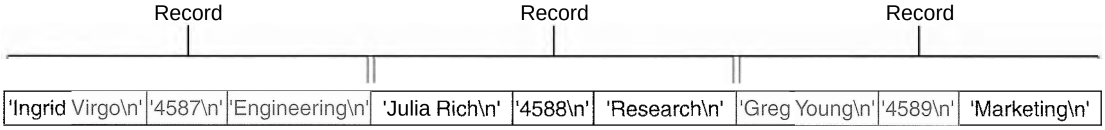
When data is written to a file, it is often organized into records and fields. A record is a complete set of data that describes one item, and a *field* is a single piece of data within a record. For example, suppose we want to store data about employees in a file. The file will contain a record for each employee. Each record will be a collection of fields, such as name, ID number, and department. This is illustrated in Figure 7-18.

Figure 7-18 Fields in a record


Record
'Ingrid Virgo\n'
4587\n'
'Engineering\n'
Name field
Department field
ID number field

Each time you write a record to a file, you write the complete set of fields that make up the record. For example, Figure 7-19 shows a file that contains three employee records. Each record consists of the employee's name, ID number, and department.

Figure 7-19 Records in a file


Record
Record
Record
'Ingrid Virgo\n'
'4587\n'
'Engineering\n'
'Julia Rich\n'
'4588\n'
'Research\n'
'Greg Young\n'
'4589\n'
'Marketing\n'

Program 7-13 shows a simple example of how employee records can be written to a file.

Program 7-13 (save_emp_records.py)

```
# This program gets employee data from the user and
# saves it as records in the employee·txt file.

def main():
    # Get the number of employee records to create.
    num_emps = input('How many employee records ' + \
                     'do you want to create? ')

    # Open a file for writing.
    emp_file = open('employees.txt', 'w')

    # Get each employee's data and write it to
    # the file.
    for count in range(1, num_emps + 1):
```

```
15      # Get the data for an employee.
16      print 'Enter data for employee #' + str(count)
17      name = raw_input('Name: ')
18      id-num = raw_input('ID number: ')
19      dept = raw_input('Department: ')
20
21      # Write the data as a record to the file.
22      emp_file.write(name + '\n')
23      emp_file.write(id_num + '\n')
24      emp_file.write(dept + '\n')
25
26      # Display a blank line.
27      print
28
29      # Close the file.
30      emp_file.close()
31      print 'Employee records written to employees.txt.'
32
33  # Call the main function.
34  main()
```

Program Output (with input shown in bold)

```
How many employee records do you want to create? 3 [Enter]
Enter the data for employee #1
Name: Ingrid Virgo [Enter]
ID number: 4587 [Enter]
Department: Engineering [Enter]

Enter the data for employee #2
Name: Julia Rich [Enter]
ID number: 4588 [Enter]
Department: Research [Enter]

Enter the data for employee #3
Name: Greg Young [Enter]
ID number: 4589 [Enter]
Department: Marketing [Enter]

Employee records written to employees.txt.
```

The statement in lines 6 and 7 prompts the user for the number of employee records that he or she wants to create. Inside the loop, in lines 17 through 19, the program gets an employee's name, ID number, and department. These three items, which together make an employee record, are written to the file in lines 22 through 24. The loop iterates once for each employee record.

When we read the data from a file that contains records, we read a complete record. Program 7-14 demonstrates how we can read the employee records in the `employee.txt` file.

Program 7-14 (read_emp_records.py)

```
# This program displays the records that are
# in the employees.txt file.

def main():
    # Open the employees.txt file.
    emp_file = open('employees.txt', 'r')

    # Read the first line from the file, which is
    # the name field of the first record.
    name = emp_file.readline()

    # If a field was read, continue processing.
    while name != '':
        # Read the ID number field.
        id-num = emp_file.readline()

        # Read the department field.
        dept = emp_file.readline()

        # Strip the newlines from the fields.
        name = name.rstrip('\n')
        id-num = id_num.rstrip('\n')
        dept = dept.rstrip('\n')

        # Display the record.
        print 'Name:', name
        print 'ID:', id-num
        print 'Dept:', dept
        print

        # Read the name field of the next record.
        name = emp_file.readline()

    # Close the file.
    emp_file.close()

# Call the main function.
main()
```

Program Output

```
Name: Ingrid Virgo
ID: 4587
Dept: Engineering
```

```
Name: Julia Rich
ID: 4588
Dept: Research

Name: Greg Young
ID: 4589
Dept: Marketing
```

This program opens the file in line 6, and then in line 10 reads the first field of the first record. This will be the first employee's name. The `while` loop in line 13 tests the value to determine whether it is an empty string. If it is not, then the loop iterates. Inside the loop, the program reads the record's second and third fields (the employee's ID number and department), and displays them. Then, in line 32 the first field of the next record (the next employee's name) is read. The loop starts over and this process continues until there are no more records to read.

Programs that store records in a file typically require more capabilities than simply writing and reading records. In the following In the Spotlight sections we will examine algorithms for adding records to a file, searching a file for specific records, modifying a record, and deleting a record.

In the Spotlight:

Adding and Displaying Records

Midnight Coffee Roasters, Inc. is a small company that imports raw coffee beans from around the world and roasts them to create a variety of gourmet coffees. Julie, the owner of the company, has asked you to write a series of programs that she can use to manage her inventory. After speaking with her, you have determined that a file is needed to keep inventory records. Each record should have two fields to hold the following data:

- Description—a string containing the name of the coffee
- Quantity in inventory—the number of pounds in inventory, as a floating-point number

Your first job is to write a program that can be used to add records to the file. Program 7-15 shows the code. Note that the output file is opened in append mode. Each time the program is executed, the new records will be added to the file's existing contents.

Program 7-15 (add_coffee_record.py)

```
# This program adds coffee inventory records to
# the coffee.txt file.

def main():
    # Create a variable to control the loop.
    another = 'y'
```

(program continues)

Program 7-15 *(continued)*

```

    # Open the coffee.txt file in append mode.
    coffee_file = open('coffee.txt', 'a')

    # Add records to the file.
    while another == 'y' or another == 'Y':
        # Get the coffee record data.
        print 'Enter the following coffee data:'
        descr = raw_input('Description: ')
        qty = input('Quantity (in pounds): ')

        # Append the data to the file.
        coffee_file.write(descr + '\n')
        coffee_file.write(str(qty) + '\n')

        # Determine whether the user wants to add
        # another record to the file.
        print 'Do you want to add another record?'
        another = raw_input('Y = yes, anything else = no: ')

    # Close the file.
    coffee_file.close()
    print 'Data appended to coffee.txt.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the following coffee data:
Description: Brazilian Dark Roast [Enter]
Quantity (in pounds): 18 [Enter]
Do you want to enter another record?
Y = yes, anything else = no: y [Enter]
Description: Sumatra Medium Roast [Enter]
Quantity (in pounds): 25 [Enter]
Do you want to enter another record?
Y = yes, anything else = no: n [Enter]
Data appended to coffee.txt.
```

Your next job is to write a program that displays all of the records in the inventory file. Program 7-16 shows the code.

Program 7-16 (show_coffee_records.py)

```
# This program displays the records in the
# coffee.txt file.
```

```

def main():
    # Open the coffee.txt file.
    coffee_file = open('coffee.txt', 'r')

    # Read the first record's description field.
    descr = coffee_file.readline()

    # Read the rest of the file.
    while descr != '':
        # Read the quantity field.
        qty = float(coffee_file.readline())

        # Strip the \n from the description.
        descr = descr.rstrip('\n')

        # Display the record.
        print 'Description:', descr
        print 'Quantity:', qty

        # Read the next description.
        descr = coffee_file.readline()

    # Close the file.
    coffee_file.close()

# Call the main function.
main()
```

Program Output

```
Description: Brazilian Dark Roast
Quantity: 18.0
Description: Sumatra Medium Roast
Quantity: 25.0
```

In the Spotlight:
Searching for a Record

Julie has been using the first two programs that you wrote for her. She now has several records stored in the `coffee.txt` file, and has asked you to write another program that she can use to search for records. She wants to be able to enter a description and see a list of all the records matching that description. Program 7-17 shows the code for the program.

Program 7-17 (search_coffee_records.py)

```
# This program allows the user to search the
# coffee.txt file for records matching a
# description.

def main():
    # Create a bool variable to use as a flag.
    found = False

    # Get the search value.
    search = raw_input('Enter a description to search for: ')

    # Open the coffee.txt file.
    coffee_file = open('coffee.txt', 'r')

    # Read the first record's description field.
    descr = coffee_file.readline()

    # Read the rest of the file.
    while descr != '':
        # Read the quantity field.
        qty = float(coffee_file.readline())

        # Strip the \n from the description.
        descr = descr.rstrip('\n')

        # Determine whether this record matches
        # the search value.
        if descr == search:
            # Display the record.
            print 'Description:', descr
            print 'Quantity:', qty
            print
            # Set the found flag to True.
            found = True

        # Read the next description.
        descr = coffee_file.readline()

    # Close the file.
    coffee_file.close()

    # If the search value was not found in the file
    # display a message.
    if not found:
        print 'That item was not found in the file.'
```

```

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a description to search for: Sumatra Medium Roast [Enter]
Description: Sumatra Medium Roast
Quantity: 25.0
```

Program Output (with input shown in bold)

```
Enter a description to search for: Mexican Altura [Enter]
That item was not found in the file.
```

In the Spotlight:
Modifying Records

Julie is very happy with the programs that you have written so far. Your next job is to write a program that she can use to modify the quantity field in an existing record. This will allow her to keep the records up to date as coffee is sold or more coffee of an existing type is added to inventory.

To modify a record in a sequential file, you must create a second temporary file. You copy all of the original file's records to the temporary file, but when you get to the record that is to be modified, you do not write its old contents to the temporary file. Instead, you write its new modified values to the temporary file. Then, you finish copying any remaining records from the original file to the temporary file.

The temporary file then takes the place of the original file. You delete the original file and rename the temporary file, giving it the name that the original file had on the computer's disk. Here is the general algorithm for your program.

```
Open the original file for input and create a temporary file for output.
Get the description of the record to be modified and the new value for the quantity.
Read the first description field from the original file.
While the description field is not empty:
    Read the quantity field.
    If this record's description field matches the description entered:
        Write the new data to the temporary file.
    Else:
        Write the existing record to the temporary file.
    Read the next description field.
Close the original file and the temporary file.
Delete the original file.
Rename the temporary file, giving it the name of the original file.
```

Notice that at the end of the algorithm you delete the original file and then rename the temporary file. The Python standard library's `os` module provides a function named `remove`, that deletes a file on the disk. You simply pass the name of the file as an argument to the function. Here is an example of how you would delete a file named `coffee.txt`:

```
remove('coffee.txt')
```

The `os` module also provides a function named rename, that renames a file. Here is an example of how you would use it to rename the file `temp.txt` to `coffee.txt`:

```
rename('temp.txt', 'coffee.txt')
```

Program 7-18 shows the code for the program.

Program 7-18 (modify_coffee_records.py)

```
# This program allows the user to modify the quantity
# in a record in the coffee.txt file.

import os # Needed for the remove and rename functions

def main():
    # Create a bool variable to use as a flag.
    found = False

    # Get the search value and the new quantity.
    search = raw_input('Enter a description to search for: ')
    new_qty = input('Enter the new quantity: ')

    # Open the original coffee.txt file.
    coffee_file = open('coffee.txt', 'r')

    # Open the temporary file.
    temp_file = open('temp.txt', 'w')

    # Read the first record's description field.
    descr = coffee_file.readline()

    # Read the rest of the file.
    while descr != '':
        # Read the quantity field.
        qty = float(coffee_file.readline())

        # Strip the \n from the description.
        descr = descr.rstrip('\n')

        # Write either this record to the temporary file,
        # or the new record if this is the one that is
        # to be modified.
        if descr == search:
            # Write the modified record to the temp file.
            temp_file.write(descr + '\n')
            temp_file.write(str(new_qty) + '\n')

            # Set the found flag to True.
            found = True
        else:
            # Write the original record to the temp file.
            temp_file.write(descr + '\n')
            temp_file.write(str(qty) + '\n')

        # Read the next description.
        descr = coffee_file.readline()

    # Close the coffee file and the temporary file.
    coffee_file.close()
    temp_file.close()

    # Delete the original coffee.txt file.
    os.remove('coffee.txt')

    # Rename the temporary file.
    os.rename('temp.txt', 'coffee.txt')

    # If the search value was not found in the file
    # display a message.
    if found:
        print 'The file has been updated.'
    else:
        print 'That item was not found in the file.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a description to search for: Brazilian Dark Roast [Enter]
Enter the new quantity: 10 [Enter]
The file has been updated.
```

In the Spotlight:

Deleting Records

Your last task is to write a program that Julie can use to delete records from the `coffee.txt` file. Like the process of modifying a record, the process of deleting a record from a sequential access file requires that you create a second temporary file. You copy all of the original file's records to the temporary file, except for the record that is to be deleted. The temporary file then takes the place of the original file. You delete the original file and rename the temporary file, giving it the name that the original file had on the computer's disk. Here is the general algorithm for your program.

```
Open the original file for input and create a temporary file for output.
Get the description of the record to be deleted.
Read the description field of the first record in the original file.
While the description is not empty:
    Read the quantity field.
    If this record's description field does not match the description entered:
        Write the record to the temporary file.
    Read the next description field.
Close the original file and the temporary file.
Delete the original file.
Rename the temporary file, giving it the name of the original file.
```

Program 7-19 shows the program's code.

Program 7-19 (delete_coffee_record.py)

```python
# This program allows the user to delete
# a record in the coffee.txt file.

import os # Needed for the remove and rename functions

def main():
    # Create a bool variable to use as a flag.
    found = False

    # Get the .
    search = raw_input('Which coffee do you want to delete? ')

    # Open the original coffee.txt file.
    coffee_file = open('coffee.txt', 'r')

    # Open the temporary file.
    temp_file = open('temp.txt', 'w')

    # Read the first record's description field.
    descr = coffee_file.readline()

    # Read the rest of the file.
    while descr != '':
        # Read the quantity field.
        qty = float(coffee_file.readline())

        # Strip the \n from the description.
        descr = descr.rstrip('\n')

        # If this is not the record to delete, then
        # write it to the temporary file.
        if descr != search:
            # Write the record to the temp file.
            temp_file.write(descr + '\n')
            temp_file.write(str(qty) + '\n')

            # Set the found flag to True.
            found = True

        # Read the next description.
        descr = coffee_file.readline()

    # Close the coffee file and the temporary file.
    coffee_file.close()
    temp_file.close()

    # Delete the original coffee.txt file.
    os.remove('coffee.txt')

    # Rename the temporary file.
    os.rename('temp.txt', 'coffee.txt')

    # If the search value was not found in the file
    # display a message.
    if found:
        print 'The file has been updated.'
    else:
        print 'That item was not found in the file.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Which coffee do you want to delete? Brazillian Dark Roast [Enter]
The file has been updated.
```

Checkpoint

7.16 What is a record? What is a field?

7.17 Describe the way that you use a temporary file in a program that modifies a record in a sequential access file.

7.18 Describe the way that you use a temporary file in a program that deletes a record from a sequential file.

7.4 Exceptions

CONCEPT: An exception is an error that occurs while a program is running, causing the program to abruptly halt. You can use the `try/except` statement to gracefully handle exceptions.

An exception is an error that occurs while a program is running. In most cases, an exception causes a program to abruptly halt. For example, look at Program 7-20. This program gets two numbers from the user and then divides the first number by the second number. In the sample running of the program, however, an exception occurred because the user entered 0 as the second number. (Division by 0 causes an exception because it is mathematically impossible.)

Program 7-20 (division.py)

```
# This program divides a number by another number.

def main():
    # Get two numbers.
    num1 = input('Enter a number: ')
    num2 = input('Enter another number: ')

    # Divide num1 by num2 and display the result.
    result = num1 / num2
    print num1, 'divided by', num2, 'is', result

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 10 [Enter]
Enter another number: 0 [Enter]
Traceback (most recent call last):
  File "C:/Python/division.py," line 13, in <module>
    main()
  File "C:/Python/division.py," line 9, in main
    result = num1 / num2
ZeroDivisionError: integer division or modulo by zero
```

The lengthy error message that is shown in the sample run is called a *traceback*. The traceback gives information regarding the line number(s) that caused the exception. (When an exception occurs, programmers say that an exception was raised.) The last line of the error message shows the name of the exception that was raised (`ZeroDivisionError`) and a brief description of the error that caused the exception to be raised (`integer division or modulo by zero`).

You can prevent many exceptions from being raised by carefully coding your program. For example, Program 7-21 shows how division by 0 can be prevented with a simple `if` statement. Rather than allowing the exception to be raised, the program tests the value of `num2`, and displays an error message if the value is 0. This is an example of gracefully avoiding an exception.

Program 7-21 (division.py)

```
# This program divides a number by another number.

def main():
    # Get two numbers.
    numl = input('Enter a number: ')
    num2 = input('Enter another number: ')

    # If num2 is not 0, divide numl by num2
    # and display the result.
    if num2 != 0:
        result = numl / num2
        print numl, 'divided by', num2, 'is', result
    else:
        print 'Cannot divide by zero.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 10 [Enter]
Enter another number: 0 [Enter]
Cannot divide by zero.
```

Some exceptions cannot be avoided regardless of how carefully you write your program. For example, look at Program 7-22. This program gets the name of a file from the user and then displays the contents of the file. The program works as long as the user enters the name of an existing file. An exception will be raised, however, if the file specified by the user does not exist. This is what happened in the sample run. (Notice in the error message that the name of the exception that occurred is `IOError`.)

Program 7-22 (display_file.py)

```
# This program displays the contents
# of a file.
```

(program continues)

Program 7-22 *(continued)*

```
 3
 4  def main():
 5      # Get the name of a file.
 6      filename = raw_input('Enter a filename: ')
 7
 8      # Open the file.
 9      infile = open(filename, 'r')
10
11      # Read the file's contents.
12      contents = infile.read()
13
14      # Display the file's contents.
15      print contents
16
17      # Close the file.
18      infile.close()
19
20  # Call the main function.
21  main()
```

Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt [Enter]
Traceback (most recent call last):
  File "C:/Python/display_file.py," line 21, in <module>
    main()
  File "C:/Python/display_file.py," line 9, in main
    infile = open(filename, 'r')
IOError: [Errno 2] No such file or directory: 'bad_file.txt'
```

Python, like most modern programming languages, allows you to write code that responds to exceptions when they are raised, and prevents the program from abruptly crashing. Such code is called an *exception handler*, and is written with the `try/except` statement. There are several ways to write a `try/except` statement, but the following general format shows the simplest variation:

```
try:
    statement
    statement
    etc.
except ExceptionName:
    statement
    statement
    etc.
```

First the key word **`try`** appears, followed by a colon. Next, a code block appears which we will refer to as the *try block*. The *try block* is one or more statements that can potentially raise an exception.

After the try block, an ***except*** clause appears. The **`except`** clause begins with the key word **`except,`** optionally followed by the name of an exception, and ending with a colon. Beginning on the next line is a block of statements that we will refer to as a handler.

When the `try/except` statement executes, the statements in the try block begin to execute. The following describes what happens next:

- If a statement in the try block raises an exception that is specified by the *`ExceptionName`* in an **`except`** clause, then the handler that immediately follows the **`except`** clause executes. Then, the program resumes execution with the statement immediately following the `try/except` statement.

  If a statement in the try block raises an exception that is *not* specified by the *`ExceptionName`* in an **`except`** clause, then the program will halt with a traceback error message.
- If the statements in the try block execute without raising an exception, then any **`except`** clauses and handlers in the statement are skipped and the program resumes execution with the statement immediately following the **`try/except`** statement.

Program 7-22 shows how we can write a `try/except` statement to gracefully respond to an **`IOError`** exception.

Program ***7-22*** (display_file.py)

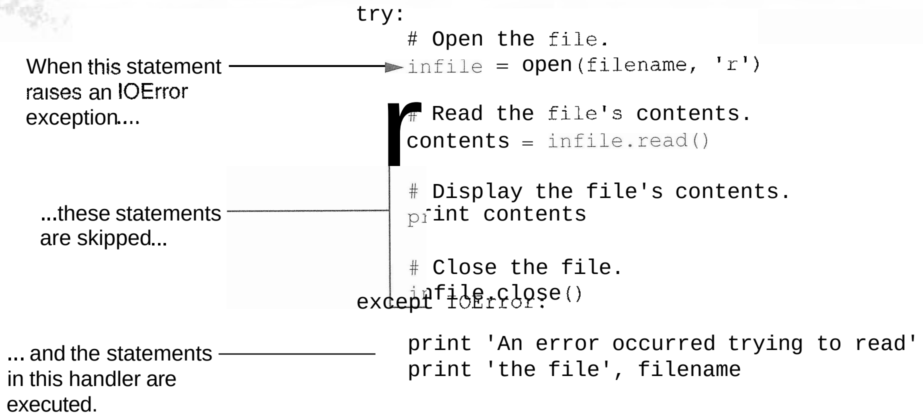
```
# This program displays the contents
# of a file.

def main():
    # Get the name of a file.
    filename = raw_input('Enter a filename: ')

    try:
        # Open the file.
        infile = open(filename, 'r')

        # Read the file's contents.
        contents = infile.read()

        # Display the file's contents.
        print contents

        # Close the file.
        infile.close()
    except IOError:
        print 'An error occurred trying to read'
        print 'the file', filename

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt [Enter]
An error occurred trying to read
the file bad_file.txt
```

Let's look at what happened in the sample run. When line 6 executed, the user entered `bad_file.txt`, which was assigned to the `filename` variable. Inside the try block, line 10 attempts to open the file `bad_file.txt`. Because this file does not exist, the statement raises an `IOError` exception. When this happens, the program exits the try block, skipping lines 11 through 19. Because the `except` clause in line 20 specifies the `IOError` exception, the program jumps to the handler that begins on line 21. Figure 7-20 illustrates this sequence of events.

Figure 7-23 Sequence of events in the `try/except` statement



```
try:
    # Open the file.
    infile = open(filename, 'r')

    # Read the file's contents.
    contents = infile.read()

    # Display the file's contents.
    print contents

    # Close the file.
    infile.close()
except IOError:
    print 'An error occurred trying to read'
    print 'the file', filename
```

When this statement raises an IOError exception....
...these statements are skipped...
... and the statements in this handler are executed.

Handling Multiple Exceptions

In many cases, the code in a try block will be capable of throwing more than one type of exception. In such a case, you need to write an `except` clause for each type of exception that you want to handle. For example, Program 7-23 reads the contents of a file named `sales_data.txt`. Each line in the file contains the sales amount for one month, and the file has several lines. Here are the contents of the file:

```
24987.62
26978.97
32589.45
31978.47
22781.76
29871.44
```

Program 7-23 reads all of the numbers from the file and adds them to an accumulator variable.

Program 7-23 (sales_report.py)

```
 1  # This program displays the total of the
 2  # amounts in the sales_data.txt file.
 3
 4  def main():
 5      # Initialize an accumulator.
 6      total = 0.0
 7
 8      try:
 9          # Open the sales_data.txt file.
10          infile = open('sales_data.txt', 'r')
11
12          # Read the values from the file and
13          # accumulate them.
14          for line in infile:
15              amount = float(line)
16              total += amount
17
18          # Close the file.
19          infile.close()
20
21          # Print the total.
22          print 'Total: $%.2f' % total
23
24      except IOError:
25          print 'An error occured trying to read the file.'
26
27      except ValueError:
28          print 'Non-numeric data found in the file.'
29
30      except:
31          print 'An error occured.'
32
33  # Call the main function.
34  main()
```

The try block contains code that can raise different types of exceptions.. For example:

- The statement in line 10 can raise an **IOError** exception if the **sales-data.txt** file does not exist. The **for** loop in line 14 can also raise an **IOError** exception if it encounters a problem reading data from the file.
- The **float** function in line 15 can raise a `ValueError` exception if the **line** variable references a string that cannot be converted to a floating-point number (an alphabetic string, for example).

Notice that the `try/except` statement has three `except` clauses:

- The `except` clause in line 24 specifies the `IOError` exception. Its handler in line 25 will execute if an `IOError` exception is raised.
- The `except` clause in line 27 specifies the `ValueError` exception. Its handler in line 28 will execute if a `ValueError` exception is raised.
- The `except` clause in line 30 does not list a specific exception. Its handler in line 31 will execute if an exception that is not handled by the other `except` clauses is raised.

In this section you've seen examples of programs that can raise `ZeroDivisionError` exceptions, `IOError` exceptions, and `ValueError` exceptions. There are many different types of exceptions that can occur in a Python program. When you are designing `try/except` statements, one way you can learn about the exceptions that you need to handle is to consult the Python documentation. It gives detailed information about each possible exception, and the types of errors that can cause them to occur.

Another way that you can learn about the exceptions that can occur in a program is through experimentation. You can run a program and deliberately perform actions that will cause errors. By watching the traceback error messages that are displayed you will see the names of the exceptions that are raised. You can then write `except` clauses to handle these exceptions.

Checkpoint

7.16 Briefly describe what an exception is.

7.17 If an exception is raised and the program does not handle it with a `try/except` statement, what happens?

7.18 What type of exception does a program raise when it tries to open a non-existent file?

7.19 What type of exception does a program raise when it uses the `float` function to convert a non-numeric string to a number?

Review Questions

Multiple Choice

1. A file that data is written to is known as a(n)
   a. input file
   b. output file
   c. sequential access file
   d. binary file
2. A file that data is read from is known as a(n)
   a. input file
   b. output file
   c. sequential access file
   d. binary file
3. Before a file can be used by a program, it must be
   a. formatted
   b. encrypted
   c. closed
   d. opened

4. When a program is finished using a file, it should do this.
   a. erase the file
   b. open the file
   c. close the file
   d. encrypt the file
5. The contents of this type of file can be viewed in an editor such as Notepad.
   a. text file
   b. binary file
   c. English file
   d. human-readable file
6. This type of file contains data that has not been converted to text.
   a. text file
   b. binary file
   c. Unicode file
   d. symbolic file
7. When working with this type of file, you access its data from the beginning of the file to the end of the file.
   a. ordered access
   b. binary access
   c. direct access
   d. sequential access
8. When working with this type of file, you can jump directly to any piece of data in the file without reading the data that comes before it.
   a. ordered access
   b. binary access
   c. direct access
   d. sequential access
9. This is a small "holding section" in memory that many systems write data to before writing the data to a file.
   a. buffer
   b. variable
   c. virtual file
   d. temporary file
10. This marks the location of the next item that will be read from a file.
    a. input position
    b. delimiter
    c. pointer
    d. read position
11. When a file is opened in this mode, data will be written at the end of the file's existing contents.
    a. output mode
    b. append mode
    c. backup mode
    d. read-only mode

12. This is a single piece of data within a record.
    a. field
    b. variable
    c. delimiter
    d. subrecord
13. When an exception is generated, it is said to have been ________.
    a. built
    b. raised
    c. caught
    d. killed
14. This is a section of code that gracefully responds to exceptions.
    a. exception generator
    b. exception manipulator
    c. exception handler
    d. exception monitor
15. You write this statement to respond to exceptions.
    a. `run/handle`
    b. `try/except`
    c. `try/handle`
    d. `attempt/except`

True or False

1. When working with a sequential access file, you can jump directly to any piece of data in the file without reading the data that comes before it.
2. When you open a file that file already exists on the disk using the `'w'` mode, the contents of the existing file will be erased.
3. The process of opening a file is only necessary with input files. Output files are automatically opened when data is written to them.
4. When an input file is opened, its read position is initially set to the first item in the file.
5. When a file that already exists is opened in append mode, the file's existing contents are erased.
6. If you do not handle an exception, it is ignored by the Python interpreter and the program continues to execute.
7. You can have more than one `except` clause in a `try/except` statement.

Short Answer

1. Describe the three steps that must be taken when a file is used by a program.
2. Why should a program close a file when it's finished using it?
3. What is a file's read position? Where is the read position when a file is first opened for reading?
4. If an existing file is opened in append mode, what happens to the file's existing contents?
5. If a file does not exist and a program attempts to open it in append mode, what happens?

Algorithm Workbench

1. Write a program that opens an output file with the filename `my_name.txt`, writes your name to the file, and then closes the file.
2. Write a program that opens the `my_name.txt` file that was created by the program in question 1, reads your name from the file, displays the name on the screen, and then closes the file.
3. Write code that does the following: opens an output file with the filename `number_list.txt`, uses a loop to write the numbers 1 through 100 to the file, and then closes the file.
4. Write code that does the following: opens the `number_list.txt` file that was created by the code you wrote in question 3, reads all of the numbers from the file and displays them, and then closes the file.
5. Modify the code that you wrote in question 4 so it adds all of the numbers read from the file and displays their total.
6. Write code that opens an output file with the filename `number_list.txt`, but does not erase the file's contents if it already exists.
7. A file exists on the disk named `students.txt`. The file contains several records, and each record contains two fields: (1) the student's name, and (2) the student's score for the final exam. Write code that deletes the record containing "John Perz" as the student name.
8. A file exists on the disk named `students.txt`. The file contains several records, and each record contains two fields: (1) the student's name, and (2) the student's score for the final exam. Write code that changes Julie Milan's score to 100.
9. What will the following code display?

```
try:
    x = float('abc123')
    print 'The conversion is complete.'
except IOError:
    print 'This code caused an IOError.'
except ValueError:
    print 'This code caused a ValueError.'
print 'The end.'
```

10. What will the following code display?

```
try:
    x = float('abc123')
    print x
except IOError:
    print 'This code caused an IOError.'
except ZeroDivisionError:
    print 'This code caused a ZeroDivisionError.'
except:
    print 'An error happened.'
print 'The end.'
```

Programming Exercises

1. File Display

Assume that a file containing a series of integers is named `numbers.txt` and exists on the computer's disk. Write a program that displays all of the numbers in the file.

2. File Head Display

Write a program that asks the user for the name of a file. The program should display only the first five lines of the file's contents. If the file contains less than five lines, it should display the file's entire contents.

3. Line Numbers

Write a program that asks the user for the name of a file. The program should display the contents of the file with each line preceded with a line number followed by a colon. The line numbering should start at 1.

4. Item Counter

Assume that a file containing a series of names (as strings) is named `names.txt` and exists on the computer's disk. Write a program that displays the number of names that are stored in the file. *(Hint: Open the file and read every string stored in it. Use a variable to keep a count of the number of items that are read from the file.)*

5. Sum of Numbers

Assume that a file containing a series of integers is named `numbers.txt` and exists on the computer's disk. Write a program that reads all of the numbers stored in the file and calculates their total.

6. Average of Numbers

Assume that a file containing a series of integers is named `numbers.txt` and exists on the computer's disk. Write a program that calculates the average of all the numbers stored in the file.

7. Exception Handing

Modify the program that you wrote for Exercise 6 so it handles the following exceptions:

- It should handle any `IOError` exceptions that are raised when the file is opened and data is read from it.
- It should handle any `ValueError` exceptions that are raised when the items that are read from the file are converted to a number.

8. Golf Scores

The Springfork Amateur Golf Club has a tournament every weekend. The club president has asked you to write two programs:

1. A program that will read each player's name and golf score as keyboard input, and then save these as records in a file named `golf.txt`. (Each record will have a field for the player's name and a field for the player's score.)
2. A program that reads the records from the `golf.txt` file and displays them.

CHAPTER

8 Working with Sequences: Strings and Lists

TOPICS

8.1 Sequences
8.2 Working with Strings
8.3 Lists

8.1 Sequences

CONCEPT: A sequence is an object that holds multiple items of data, stored one after the other. You can perform operations on a sequence, to examine and manipulate the items stored in it.

A sequence is an object that contains multiple items of data. The items that are in a sequence are stored one after the other. Python provides various ways to perform operations on the items that are stored in a sequence.

There are several different types of sequence objects in Python. In this chapter we will look at two of the fundamental sequence types: strings and lists. You've worked with strings already, and you know that a string is a sequence of characters. Lists are sequences that can hold various types of data. We will explore some of the operations that you may perform on these sequences, including ways to access and manipulate their contents.

8.2 Working with Strings

CONCEPT: Python provides several ways to access the individual characters in a string. Strings also have methods that allow you to perform operations on them.

Many of the programs that you have written so far have worked with strings, but only in a limited way. The operations that you have performed with strings so far have primarily involved only input and output. For example, you have read strings as input from the keyboard and from files, and sent strings as output to the screen and to files.

There are many types of programs that not only read strings as input and write strings as output, but also perform operations on strings. Word processing programs, for example, manipulate large amounts of text, and thus work extensively with strings. Email programs and search engines are other examples of programs that perform operations on strings.

Python provides a wide variety of tools and programming techniques that you can use to examine and manipulate strings. We will look at many of these in this section.

Accessing the Individual Characters in a String

Some programming tasks require that you access the individual characters in a string. For example, you are probably familiar with websites that require you to set up a password. For security reasons, many sites require that your password have at least one uppercase letter, at least one lowercase letter, and at least one digit. When you set up your password, a program examines each character to ensure that the password meets these qualifications. (Later in this chapter you will see an example of a program that does this sort of thing.) In this section we will look at two techniques that you can use in Python to access the individual characters in a string: using the `for` loop, and indexing.

Iterating Over a String with the `for` Loop

One of the easiest ways to access the individual characters in a string is to use the `for` loop. Here is the general format:

```
for variable in string:
    statement
    statement
    etc.
```

In the general format, *`variable`* is the name of a variable and `string` is either a string literal or a variable that references a string. Each time the loop iterates, *`variable`* will reference a copy of a character in `string`, beginning with the first character. We say that the loop iterates over the characters in the string. Here is an example:

```
name = 'Juliet'
for ch in name:
    print ch
```

The name variable references a string with six characters, so this loop will iterate six times. The first time the loop iterates, the `ch` variable will reference `'J'`, the second time the loop iterates the `ch` variable will reference `'u'`, and so forth. This is illustrated in Figure 8-1. When the code executes, it will display the following:

```
J
u

i
e
t
```

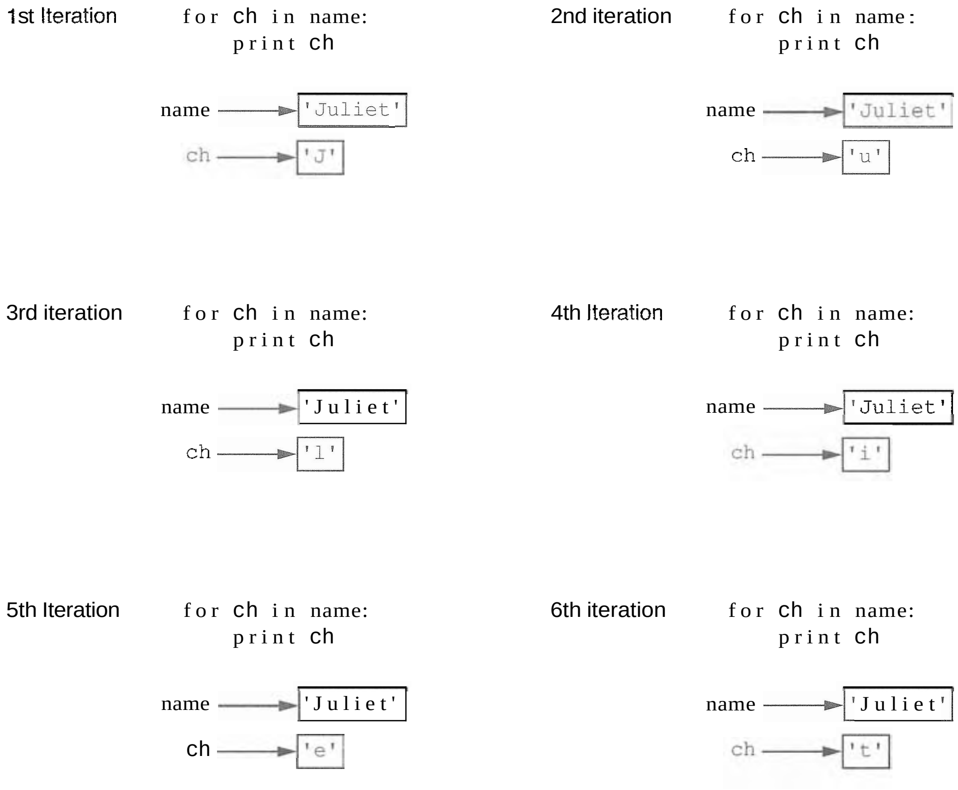


Figure 8-1 Iterating over the string `'Juliet'`

1st Iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
'J'

2nd iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
'u'

3rd iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
'l'

4th Iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
'i'

5th Iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
'e'

6th iteration

```
for ch in name:
    print ch
```

name
'Juliet'
ch
't'

NOTE: Figure 8-1 illustrates how the `ch` variable references a copy of a character from the string as the loop iterates. If we change the value that `ch` references in the loop, it has no effect on the string referenced by name. To demonstrate, look at the following:

```
1 name = 'Juliet'
2 for ch in name:
3     ch = 'X'
4     print name
```

The statement in line **3** merely reassigns the `ch` variable to a different value each time the loop iterates. It has no effect on the string `'Juliet'` that is referenced by name, and it has no effect on the number of times the loop iterates. When this code executes, the statement in line 4 will print:

```
Juliet
```

Program 8-1 shows another example. This program asks the user to enter a string. It then uses a `for` loop to iterate over the string, counting the number of times that the letter T (uppercase or lowercase) appears.

Program 8-1 **(count_Ts.py)**

```
# This program counts the number times
# the letter T (uppercase or lowercase)
# appears in a string.

def main():
    # Create a variable to use to hold the count.
    # The variable must start with 0.
    count = 0

    # Get a string from the user.
    my-string = raw_input('Enter a sentence: ')

    # Count the Ts.
    for ch in my-string:
        if ch == 'T' or ch == 't':
            count += 1

    # Print the result.
    print 'The letter T appears', count, 'times.'

# Call the main function.
main()
```

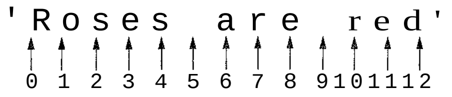
Program Output (with input shown in bold)

```
Enter a sentence: Today we sold twenty-two toys. [Enter]
The letter T appears 5 times.
```

Indexing

Another way that you can access the individual characters in a string is with an index. Each character in a string has an ***index*** which specifies its position in the string. Indexing starts at 0, so the index of the first character is 0, the index of the second character is 1, and so forth. The index of the last character in a string is 1 less than the number of characters in the string. Figure 8-2 shows the indexes for each character in the string `'Roses are red'`. The string has 13 characters, so the character indexes range from 0 through 12.

Figure 8-2 String indexes

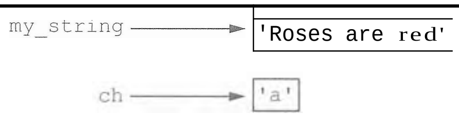


You can use an index to retrieve a copy of an individual character in a string, as shown here:

```
my-string = 'Roses are red'
ch = my_string[6]
```

The expression `my_string[6]` in the second statement returns a copy of the character at index 6 in `my_string`. After this statement executes, `ch` will reference `'a'` as shown in Figure 8-3.

Figure 8-3 Getting a copy of a character from a string



Here is another example:

```
my-string = 'Roses are red'
print my_string[0], my_string[6], my_string[10]
```

This code will print the following:

```
R a r
```

You can also use negative numbers as indexes, to identify character positions relative to the end of the string. The Python interpreter adds negative indexes to the length of the string to determine the character position. The index –1 identifies the last character in a string, –2 identifies the next to last character, and so forth. The following code shows an example:

```
my-string = 'Roses are red'
print my-string[-11, my-string[-21, my-string[-131
```

This code will print the following:

```
d e R
```

`IndexError` Exceptions

An `IndexError` exception will occur if you try to use an index that is out of range for a particular string. For example, the string `'Boston'` has 6 characters, so the valid indexes are 0 through 5. (The valid negative indexes are –1 through –6.) The following is an example of code that causes an `IndexError` exception.

```
city = 'Boston'
print city[6]
```

This type of error is most likely to happen when a loop incorrectly iterates beyond the end of a string, as shown here:

```
city = 'Boston'
index = 0
while index < 7:
    print city[index]
    index += 1
```

The last time that this loop iterates, the `index` variable will be assigned the value 6, which is an invalid index for the string `'Boston'`. As a result, the `print` statement will cause an `IndexError` exception to be raised.

The `len` Function

Python has a built-in function named `len` that returns the length of a sequence, such as a string. The following code demonstrates:

```
city = 'Boston'
size = len(city)
```

The second statement calls the `len` function, passing the `city` variable as an argument. The function returns the value 6, which is the length of the string `'Boston'`. This value is assigned to the `size` variable.

The `len` function is especially useful to prevent loops from iterating beyond the end of a string, as shown here:

```
city = 'Boston'
index = 0
while index < len(city):
    print city[index]
    index += 1
```

Notice that the loop iterates as long as `index` is less *than* the length of the string. This is because the index of the last character in a string is always 1 less than the length of the string.

Strings Are Immutable

In Python, strings are immutable, which means that once they are created, they cannot be changed. Some operations, such as concatenation, give the impression that they modify strings, but in reality they do not. For example, look at Program 8-2.

Program 8-2 (concatenate.py)

```
# This program concatenates strings.

def main():
    name = 'Carmen'
    print 'The name is', name
    name = name + ' Brown'
    print 'Now the name is', name

# Call the main function.
main()
```

Program Output

```
The name is Carmen
Now the name is Carmen Brown
```

The statement in line 4 assigns the string `'Carmen'` to the name variable, as shown in Figure 8-4. The statement in line 6 concatenates the string `' Brown'` to the string `'Carmen'` and assigns the result to the name variable, as shown in Figure 8-5. As you can see from the figure, the original string `'Carmen'` is not modified. Instead, a new string containing 'Carmen Brown' is created and assigned to the name variable. (The original string, `'Carmen'` is no longer usable because no variable references it. The Python interpreter will eventually remove the unusable string from memory.)

Figure 8-4 *The string 'Carmen' assigned to* name

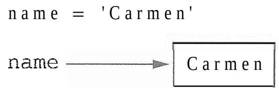
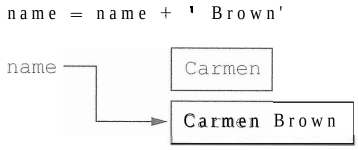


Figure 8-5 *The string 'Carmen Brown' assigned to* name



Because strings are immutable, you cannot use an expression in the form *`string[index]`* on the left side of an assignment operator. For example, the following code will cause an error:

```
# Assign 'Bill' to friend.
friend = 'Bill'
# Can we change the first character to 'J'?
friend[0] = 'J'      # No, this will cause an error!
```

The last statement in this code will raise an exception because it attempts to change the value of the first character in the string `'Bill'`.

String Slicing

A *slice* is a span of items that are taken from a sequence. When you take a slice from a string, you get a span of characters from within the string. String slices are also called *substrings*.

To get a slice of a string, you write an expression in the following general format:

```
string[start : end]
```

In the general format, `start` is the index of the first character in the slice, and `end` is the index marking the end of the slice. The expression will return a string containing a copy of the characters from `start` up to (but not including) `end`. For example, suppose we have the following:

```
full_name = 'Patty Lynn Smith'
middle_name = full_name[6:10]
```

The second statement assigns the string `'Lynn'` to the `middle_name` variable. If you leave out the *start* index in a slicing expression, Python uses 0 as the starting index. Here is an example:

```
full-name = 'Patty Lynn Smith'
first-name = full_name[:5]
```

The second statement assigns the string `'Lynn'` to `first_name`. If you leave out the *end* index in a slicing expression, Python uses the length of the string as the *end* index. Here is an example:

```
full-name = 'Patty Lynn Smith'
last-name = full_name[11:]
```

The second statement assigns the string `'Smith'` to `first_name`. What do you think the following code will assign to the `my_string` variable?

```
full_name = 'Patty Lynn Smith'
my_string = full-name[:]
```

The second statement assigns the entire string `'Patty Lynn Smith'` to `my_string`. The statement is equivalent to:

```
my-string = full_name[0 : len(full_name)]
```

The slicing examples we have seen so far get slices of consecutive characters from strings. Slicing expressions can also have step value, which can cause characters to be skipped in the string. Here is an example of code that uses a slicing expression with a step value:

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print letters[0:26:2]
```

The third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second character from the specified range in the string. The code will print the following:

```
ACEGIKMOQSUWY
```

You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the string. Here is an example:

```
full_name = 'Patty Lynn Smith'
last-name = full-name[-5:]
```

Recall that Python adds a negative index to the length of a string to get the position referenced by that index. The second statement in this code assigns the string `'Smith'` to the `last_name` variable.

NOTE: Invalid indexes do not cause slicing expressions to raise an exception. For example:

- If the *end* index specifies a position beyond the end of the string, Python will use the length of the string instead.
- If the *start* index specifies a position before the beginning of the string, Python will use 0 instead.
- If the *start* index is greater than the *end* index, the slicing expression will return an empty string.

In the Spotlight:

Extracting Characters from a String

At a university, each student is assigned a system login name, which the student uses to log into the campus computer system. As part of your internship with the university's Information Technology department, you have been asked to write the code that generates system login names for students. You will use the following algorithm to generate a login name:

1. *Get the first three characters of the student's first name. (If the first name is less than three characters in length, use the entire first name.)*
2. *Get the first three characters of the student's last name. (If the last name is less than three characters in length, use the entire last name.)*
3. *Get the last three characters of the student's ID number. (If the ID number is less than three characters in length, use the entire ID number.)*
4. *Concatenate the three sets of characters to generate the login name.*

For example, if a student's name is Amada Spencer, and her ID number is ENG6721, her login name would be AmaSpe721. You decide to write a function named `get_login_name` that accepts a student's first name, last name, and ID number as arguments, and returns the student's login name as a string. You will save the function in a module named `login.py`. This module can then be imported into any Python program that needs to generate a login name. Program 8-3 shows the code for the `login.py` module.

Program 8-3 (login.py)

```
# The get_login_name function accepts a first name,
# last name, and ID number as arguments. It returns
# a system login name.

def get_login_name(first, last, idnumber):
    # Get the first three letters of the first name.
    # If the name is less than 3 characters, the
    # slice will return the entire first name.
    set1 = first[0 : 3]

    # Get the first three letters of the last name.
    # If the name is less than 3 characters, the
    # slice will return the entire last name.
    set2 = last[0 : 3]

    # Get the last three characters of the student ID.
    # If the ID number is less than 3 characters, the
    # slice will return the entire ID number.
    set3 = idnumber[-3 :]

    # Put the sets of characters together.
```

(program continues)

Program 8-3 *(continued)*

```
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
```

The `get_login_name` function accepts three string arguments: a first name, a last name, and an ID number. The statement in line 9 uses a slicing expression to get the first three characters of the string referenced by `first`, and assigns those characters, as a string, to the `set1` variable. If the string referenced by `first` is less than 3 characters long, then the value 3 will be an invalid ending index. If this is the case, Python will use the length of the string as the ending index, and the slicing expression will return the entire string.

The statement in line 14 uses a slicing expression to get the first three characters of the string referenced by `last`, and assigns those characters, as a string, to the `set2` variable. The entire string referenced by `last` will be returned if it is less than 3 characters.

The statement in line 19 uses a slicing expression to get the last three characters of the string referenced by `idnumber`, and assigns those characters, as a string, to the `set3` variable. If the string referenced by `idnumber` is less than 3 characters, then the value −3 will be an invalid starting index. If this is the case, Python will use 0 as the starting index.

The statement in line 22 assigns the concatenation of `set1`, `set2`, and `set3` to the `login_name` variable. The variable is returned in line 25. Program 8-4 shows a demonstration of the function.

Program 8-4 (generate_login.py)

```
# This program gets the user's first name, last name, and
# student ID number. Using this data it generates a
# system login name.

import login

def main():
    # Get the user's first name, last name, and ID number.
    first = raw_input('Enter your first name: ')
    last = raw_input('Enter your last name: ')
    idnumber = raw_input('Enter your student ID number: ')

    # Get the login name.
    print 'Your system login name is:'
    print login.get_login_name(first, last, idnumber)

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your first name: Holly [Enter]
Enter your last name: Gaddis [Enter]
Enter your student ID number: CSC34899 [Enter]
Your system login name is:
HolGad899
```

Program Output (with input shown in bold)

```
Enter your first name: Jo [Enter]
Enter your last name: Cusimano [Enter]
Enter your student ID number: BIO4497 [Enter]
Your system login name is:
JoCus497
```

Testing Strings with `in` and `not in`

In Python you can use the `in` operator to determine whether one string is contained in another string. Here is the general format of an expression using the `in` operator with two strings:

```
string1 in string2
```

string1 and *string2* can be either string literals or variables referencing strings. The expression returns true if *string1* is found in *string2*. For example, look at the following code:

```
text = 'Four score and seven years ago'
if 'seven' in text:
    print 'The string "seven" was found.'
else:
    print 'The string "seven" was not found.'
```

This code determines whether the string `'Four score and seven years ago'` contains the string `'seven'`. If we run this code it will display:

```
The string "seven" was found.
```

You can use the `not in` operator to determine whether one string is *not* contained in another string. Here is an example:

```
names = 'Bill Joanne Susan Chris Juan Katie'
if 'Pierre' not in names:
    print 'Pierre was not found.'
else:
    print 'Pierre was found.'
```

If we run this code it will display:

```
Pierre was not found.
```

String Methods

Recall from Chapter 7 that a method is a function that belongs to an object, and performs some operation on that object. Strings in Python have numerous methods.[1] In this section we will discuss several string methods for performing the following types of operations:

- Testing the values of strings
- Performing various modifications
- Searching for substrings and replacing sequences of characters

Here is the general format of a string method call:

```
stringvar.method(arguments)
```

In the general format, *stringvar* is a variable that references a string, *method* is the name of the method that is being called, and *arguments* is one or more arguments being passed to the method. Let's look at some examples.

String Testing Methods

The string methods shown in Table 8-1 test a string for specific characteristics. For example, the `isdigit` method returns true if the string contains only numeric digits. Otherwise, it returns false. Here is an example:

```
stringl = '1200'
if string1.isdigit():
    print stringl, 'contains only digits.'
else:
    print stringl, 'contains characters other than digits.'
```

This code will display

```
1200 contains only digits.
```

Here is another example:

```
string2 = '123abc'
if string2.isdigit():
    print string2, 'contains only digits.'
else:
    print string2, 'contains characters other than digits.'
```

This code will display

```
123abc contains characters other 'than digits.
```

Program 8-5 demonstrates several of the string testing methods. It asks the user to enter a string, and then displays various messages about the string, depending on the return value of the methods.

[1] We do not cover all of the string methods in this book. For a comprehensive list of string methods, see the Python documentation at www.python.org.

Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Program 8-5 (string_test.py)

```
# This program demonstrates several string testing methods.

def main():
    # Get a string from the user.
    user_string = raw_input('Enter a string: ')

    print 'This is what I found about that string:'

    # Test the string.
    if user_string.isalnum():
        print 'The string is alphanumeric.'
    if user_string.isdigit():
        print 'The string contains only digits.'
    if user_string.isalpha():
        print 'The string contains only alphabetic characters.'
    if user_string.isspace():
        print 'The string contains only whitespace characters.'
    if user_string.islower():
        print 'The letters in the string are all lowercase.'
    if user_string.isupper():
        print 'The letters in the string are all uppercase.'

# Call the string.
main()
```

Program Output (with input shown in bold)

```
Enter a string: abc [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only alphabetic characters.
The letters in the string are all lowercase.
```

Program Output (with input shown in bold)

```
Enter a string: 123 [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only digits.
```

Program Output (with input shown in bold)

```
Enter a string: 123ABC [Enter]
This is what I found about that string:
The string is alphanumeric.
The letters in the string are all uppercase.
```

Modification Methods

Although strings are immutable, meaning they cannot be modified, they do have a number of methods that return modified versions of themselves. Table 8-2 lists several of these methods.

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The ***char*** argument is a string containing a character. Returns a copy of the string with all instances of ***char*** that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The ***char*** argument is a string containing a character. The method returns a copy of the string with all instances of ***char*** .that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of ***char*** that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

For example, the `lower` method returns a copy of a string with all of its alphabetic letters converted to lowercase. Here is an example:

```
letters = 'WXYZ'
print letters, letters.lower()
```

This code will print:

```
WXYZ wxyz
```

The `upper` method returns a copy of a string with all of its alphabetic letters converted to uppercase. Here is an example:

```
letters = 'abcd'
print letters, letters.upper()
```

This'code will print:

```
abcd ABCD
```

The `lower` and `upper` methods are useful for making case-insensitive string comparisons. String comparisons are *case-sensitive,* which means that the uppercase characters are distinguished from the lowercase characters. For example, in a case-sensitive comparison, the string `'abc'` is not considered the same as the string `'ABC'` or the string `'Abc'` because the case of the characters are different. Sometimes it is more convenient to perform a *case-insensitive* comparison, in which the case of the characters is ignored. In a case-insensitive comparison, the string `'abc'` is considered the same as `'ABC'` and `'Abc'`.

For example, look at the following code:

```
again = 'y'
while again.lower() == 'y':
    print 'Hello'
    print 'Do you want to see that again?'
    again = raw-input('y = yes, anything else = no: ')
```

Notice that the last statement in the loop asks the user to enter `y` to see the message displayed again. The loop iterates as long as the expression `again.lower() == 'y'` is true. The expression will be true if the `again` variable references either `'y'` or `'Y'`.

Similar results can be achieved by using the `upper` method, as shown here:

```
again = 'y'
while again.upper() == 'Y':
    print 'Hello'
    print 'Do you want to see that again?'
    again = raw-input('y = yes, anything else = no: ')
```

Searching and Replacing

Programs commonly need to search for substrings, or strings that appear within other strings. For example, suppose you have a document opened in your word processor, and you need to search for a word that appears somewhere in it. The word that you are searching for is a substring that appears inside a larger string, the document.

Table 8-3 lists some of the Python string methods that search for substrings, as well as a method that replaces the occurrences of a substring with another string.

Table 8-3 Search and replace methods

Method	Description
`endswith(substring)`	The `substring` argument is a string. The method returns true if the string ends with `substring`.
`find(substring)`	The `substring` argument is a string. The method returns the lowest index in the string where `substring` is found. If `substring` is not found, the method returns −1.
`replace(old, new)`	The `old` and `new` arguments are both strings. The method returns a copy of the string with all instances of `old` replaced by `new`.
`startswith(substring)`	The `substring` argument is a string. The method returns true if the string starts with `substring`.

The `endswith` method determines whether a string ends with a specified substring. Here is an example:

```
filename = raw_input('Enter the filename: ')
if filename.endswith('.txt'):
    print 'That is the name of a text file.'
elif filename.endswith('.py'):
    print 'That is the name of a Python source file.'
elif filename.endswith('.doc'):
    print 'That is the name of a word processing document
else:
    print 'Unknown file type.'
```

The `startswith` method works like the `endswith` method, but determines whether a string begins with a specified substring.

The `find` method searches for a specified substring within a string. The method returns the lowest index of the substring, if it is found. If the substring is not found, the method returns −1. Here is an example:

```
string = 'Four score and seven years ago'
position = string.find('seven')
if position != -1:
    print 'The word "seven" was found at index', position
else:
    print 'The word "seven" was not found.'
```

This code will display:

```
The word "seven" was found at index 15
```

The `replace` method returns a copy of a string, where every occurrence of a specified substring has been replaced with another string. For example, look at the following code:

```
string = 'Four score and seven years ago'
new-string = string.replace('years', 'days')
print new-string
```

This code will display:

```
Four score and seven days ago
```

In the Spotlight:

Validating the Characters in a Password

At the university, passwords for the campus computer system must meet the following requirements:

- The password must be at least 7 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one lowercase letter.
- It must contain at least one numeric digit.

When a student sets up his or her password, the password must be validated to ensure it meets these requirements. You have been asked to write the code that performs this validation. You decide to write a function named `valid_password` that accepts the password as an argument and returns either true or false, to indicate whether it is valid. Here is the algorithm for the function, in pseudocode:

```
valid_password function:
   Set the correct_length variable to false
   Set the has_uppercase variable to false
   Set the has_lowercase variable to false
   Set the has_digit variable to false
   If the password's length is 7 characters or greater:
      Set the correct_length variable to true
      for each character in the password:
         if the character is an uppercase letter:
            Set the has_uppercase variable to true
         if the character is a lowercase letter:
            Set the has_lowercase variable to true
         if the character is a digit:
            Set the has_digit variable to true

   If correct_length and has_uppercase and has_lowercase and has_digit:
      Set the is_valid variable to true
   else:
      Set the is_valid variable to false

   Return the is_valid variable
```

Earlier (in the previous In the Spotlight section) you created a function named `get_login_name`, and stored that function in the `login` module. Because the `valid_password` function's purpose is related to the task of creating a student's login account, you decide to store the `valid_password` function in the `login` module as well. Program 8-6 shows the login module with the `valid_password` function added to it. The function begins at line 34.

Program 8-6 (login.py)

```
 1  # The get_login_name function accepts a first name,
 2  # last name, and ID number as arguments. It returns
 3  # a system login name.
 4
 5  def get_login_name(first, last, idnumber):
 6      # Get the first three letters of the first name.
 7      # If the name is less than 3 characters, the
 8      # slice will return the entire first name.
 9      set1 = first[0 : 3]
10
11      # Get the first three letters of the last name.
12      # If the name is less than 3 characters, the
13      # slice will return the entire last name.
14      set2 = last[0 : 3]
15
16      # Get the last three characters of the student ID.
17      # If the ID number is less than 3 characters, the
18      # slice will return the entire ID number.
19      set3 = idnumber[-3 :]
20
21      # Put the sets of characters together.
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
26
27  # The valid_password function accepts a password as
28  # an argument and returns either true or false to
29  # indicate whether the password is valid. A valid
30  # password must be at least 7 characters in length,
31  # have at least one uppercase letter, one lowercase
32  # letter, and one digit.
33
34  def valid_password(password):
35      # Set the Boolean variables to false.
36      correct_length = False
37      has_uppercase = False
38      has_lowercase = False
39      has_digit = False
```

```

    # Begin the validation. Start by testing the
    # password's length.
    if len(password) >= 7:
        correct_length = True

        # Test each character and set the
        # appropriate flag when a required
        # character is found.
        for ch in password:
            if ch.isupper():
                has_uppercase = True
            if ch.islower():
                has_lowercase = True
            if ch.isdigit():
                has_digit = True

    # Determine whether all of the requirements
    # are met. If they are, set is_valid to true.
    # Otherwise, set is_valid to false.
    if correct_length and has_uppercase and \
        has_lowercase and has_digit:
        is_valid = True
    else:
        is_valid = False

    # Return the is_valid variable.
    return is_valid
```

Program 8-7 imports the login module and demonstrates the `valid_password` function.

Program 8-7 (validate_password.py)

```
# This program gets a password from the user and
# validates it.

import login

def main():
    # Get a password from the user.
    password = raw_input('Enter your password: ')

    # Validate the password.
    while not login.valid_password(password):
        print 'That password is not valid.'
        password = raw_input('Enter your password: ')

```

(program continues)

Program 8-7 *(continued)*

```
    print 'That is a valid password.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your password: bozo [Enter]
That password is not valid.
Enter your password: kangaroo [Enter]
That password is not valid.
Enter your password: Tiger9 [Enter]
That password is not valid.
Enter your password: Leopard6 [Enter]
That is a valid password.
```

The Repetition Operator

You learned in Chapter 2 that the * symbol multiplies two numbers. However, when the operand on the left side of the * symbol is a string and the operand on the right side is an integer, it becomes the *repetition operator*. Here is the general format:

```
string-to-copy * n
```

The repetition operator creates a string that contains *n* repeated copies of *string_to_copy*. Here is an example:

```
my-string = 'w' * 5
```

After this statement executes, `my_string` will reference the string `'wwwww'`. Here is another example:

```
print 'Hello' * 5
```

This statement will print:

```
HelloHelloHelloHelloHellc
```

Program 8-8 demonstrates the repetition operator.

Program 8-8 (repetition_operator.py)

```
# This program demonstrates the repetition operator.

def main():
    # Print 9 rows increasing in length.
    for count in range(1, 10):
        print 'Z' * count

    # Print 9 rows decreasing in length.
    for count in range(8, 0, -1):
        print 'Z' * count

# Call the main function.
main()
```

Program Output

```
Z
ZZ
ZZZ
ZZZZ
ZZZZZ
ZZZZZZ
ZZZZZZZ
ZZZZZZZZ
ZZZZZZZZZ
ZZZZZZZZ
ZZZZZZZ
ZZZZZZ
ZZZZZ
ZZZZ
ZZZ
ZZ
Z
```

Checkpoint

8.1 Assume the variable name references a string. Write a `for` loop that prints each character in the string.

8.2 What is the index of the first character in a string?

8.3 If a string has 10 characters, what is the index of the last character?

8.4 What happens if you try to use an invalid index to access a character in a string?

8.5 How do you find the length of a string?

8.6 What is wrong with the following code?

```
animal = 'Tiger'
animal[0] = 'L'
```

8.7 What will the following code display?

```
mystring = 'abcdefg'
print mystring[2:5]
```

8.8 What will the following code display?

```
mystring = 'abcdefg'
print mystring[3:]
```

8.9 What will the following code display?

```
mystring = 'abcdefg'
print mystring[:3]
```

8.10 What will the following code display?

```
mystring = 'abcdefg'
print mystring[:]
```

8.11 Write code using the `in` operator that determines whether `'d'` is in `mystring`.

8.12 Assume the variable `big` references a string. Write a statement that converts the string it references to lowercase, and assigns the converted string to the variable `little`.

8.13 Write an `if` statement that displays "Digit" if the string referenced by the variable `ch` contains a numeric digit. Otherwise, it should display "No digit."

8.14 What is the output of the following code?

```
ch = 'a'
ch2 = ch.upper()
print ch, ch2
```

8.15 Write a loop that asks the user "Do you want to repeat the program or quit? (R/Q)". The loop should repeat until the user has entered an R or Q (either uppercase or lowercase).

8.16 What will the following code display?

```
var = '$'
print var.upper()
```

8.17 Write a loop that counts the number of uppercase characters that appear in the string referenced by the variable `mystring`.

8.3 Lists

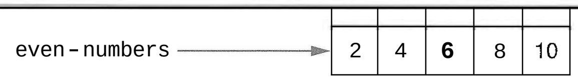
CONCEPT: **A** list is an object that contains multiple data items. Lists are mutable, which means that their contents can be changed during a program's execution. Lists are dynamic data structures, meaning that items may be added to them or removed from them. You can use indexing, slicing, and various methods to work with lists in a program.

A list is an object that contains multiple data items. Each item that is stored in a list is called an *element*. Here is a statement that creates a list of integers:

```
even-numbers = [2, 4, 6, 8, 10]
```

The items that are enclosed in brackets and separated by commas are the list elements. After this statement executes, the variable `even—numbers` will reference the list, as shown in Figure 8-6.

Figure 8-6 A list of integers

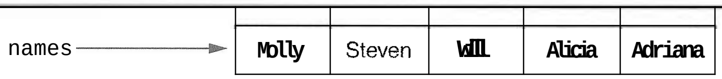


The following is another example:

```
names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']
```

This statement creates a list of five strings. After the statement executes, the name variable will reference the list as shown in Figure 8-7.

Figure 8-7 A list of strings



You can use the `print` statement to display an entire list, as shown here:

```
numbers = [5, 10, 15, 20]
print numbers
```

When the `print` statement executes, it will display the elements of the list like this:

```
[5, 10, 15, 20]
```

You can use the `range` function, which was introduced in Chapter 5, to generate a list of integers. Here is an example:

```
numbers = range(5)
```

In this statement the `range` function will return a list of integers in the range of 0 up to (but not including) 5. This statement will assign the list `[0, 1, 2, 3, 4]` to the `numbers` variable. Here is another example:

```
numbers = range(1, 10, 2)
```

Recall from Chapter 5 that when you pass three arguments to the `range` function, the first argument is the list's starting value, the second argument is the list's ending limit, and the third argument is the step value. This statement will assign the list `[1, 3, 5, 7, 9]` to the `numbers` variable.

You can use the repetition operator (`*`) to easily create a list with a specific number of elements, each with the same value. Here is an example:

```
numbers = [0] * 5
```

This statement will create a list with five elements, with each element holding the value 0. This statement will assign the list `[0, 0, 0, 0, 0]` to the `numbers` variable.

NOTE: Most programming languages allow you to create sequence structures known as *arrays*, which are similar to lists, but are much more limited in their capabilities. You cannot create traditional arrays in Python because lists serve the same purpose and provide many more built-in capabilities.

Iterating Over a List with the `for` Loop

In Section 8.1 we discussed techniques for accessing the individual characters in a string. Many of the same programming techniques also apply to lists. For example, you can iterate over a list with the `for` loop, as shown here:

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print n
```

If we run this code, it will print:

```
99
100
101
102
```

Indexing

Indexing works with lists just as it does with strings. Each element in a list has an index which specifies its position in the list. Indexing starts at 0, so the index of the first element is 0, the index of the second element is 1, and so forth. The index of the last element in a list is 1 less than the number of elements in the list.

For example, the following statement creates a list with 4 elements:

```
my-list = [10, 20, 30, 40]
```

The indexes of the elements in this list are 0, 1, 2, and 3. We can print the elements of the list with the following statement:

```
print my_list[0], my_list[1], my_list[2], my_list[3]
```

The following loop also prints the elements of the list:

```
index = 0
while index < 4:
     print my_list[index]
     index += 1
```

You can also use negative indexes with lists, to identify element positions relative to the end of the list. The Python interpreter adds negative indexes to the length of the list to determine the element position. The index –1 identifies the last element in a list, –2 identifies the next to last element, and so forth. The following code shows an example:

```
my-list = [10, 20, 30, 40]
print my-list[-11, my-list[-21, my-list[-31, my-list[-41
```

This `print` statement will display:

```
40 30 20 10
```

An `IndexError` exception will be raised if you use an invalid index with a list. For example, look at the following code:

```
# This code will cause an IndexError exception.
my-list = [10, 20, 30, 40]
```

```
index = 0
while index < 5:
    print my_list[index]
    index += 1
```

The last time that this loop iterates, the `index` variable will be assigned the value 5, which is an invalid index for the list. As a result, the `print` statement will cause an `IndexError` exception to be raised.

The `len` function that you learned about in the previous section can be used with lists as well as strings. When you pass a list as an argument, the `len` function returns the number of elements in the list. The previously shown code, which raises an `IndexError` exception, can be modified as follows to prevent the exception:

```
my_list = [10, 20, 30, 40]
index = 0
while index < len(my_list):
    print my_list[index]
    index += 1
```

Slicing

Slicing operations work with lists just as they do with strings. For example, suppose we create the following list:

```
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday', 'Saturday']
```

The following statement uses a slicing expression to get the elements from indexes 2 up to, but not including, 5:

```
mid-days = days[2:5]
```

After this statement executes the `mid_days` variable will reference the following list:

```
['Tuesday', 'Wednesday', 'Thursday'].
```

Finding Items in a List with `in` and `not in`

You can use the `in` operator to determine whether an item is contained in a list. Program 8-9 shows an example.

Program 8-9 (in_list.py)

```
# This program demonstrates the in operator
# used with a list.

def main():
    # Create a list of product numbers.
    prod-nums = ['V475', 'F987', 'Q143', 'R688']
```

(program continues)

Program 8-9 *(continued)*

```
 7
 8       # Get a product number to search for.
 9       search = raw_input('Enter a product number: ')
10
11       # Determine whether the product number is in the list.
12       if search in prod-nums:
13           print search, 'was found in the list.'
14       else:
15           print search, 'was not found in the list.'
16
17   # Call the main function.
18   main()
```

Program Output (with input shown in bold)

```
Enter a product number: Q143 [Enter]
Q143 was found in the list.
```

Program Output (with input shown in bold)

```
Enter a product number: B000 [Enter]
B000 was not found in the list.
```

The program gets a product number from the user in line 9 and assigns it to the `search` variable. The `if` statement in line 12 determines whether `search` is in the prod—nums list.

You can use the `not in` operator to determine whether an item is *not* in a list. Here is an example:

```
if search not in prod-nums:
    print search, 'was not found in the list.'
else:
    print search, 'was found in the list.'
```

Lists Are Mutable

Unlike strings, lists in Python are mutable, which means their elements can be changed. Consequently, an expression in the form `list[index]` can appear on the left side of an assignment operator. The following code shows an example:

```
1 numbers = [1, 2, 3, 4, 5]
2 print numbers
3 numbers[0] = 99
4 print numbers
```

The statement in line 2 will display:

```
[1, 2, 3, 4, 5]
```

The statement in line **3** assigns 99 to `numbers[0]`. This changes the first value in the list to 99. When the statement in line 4 executes it will display:

```
[99, 2, 3, 4, 5]
```

When you use an indexing expression to assign a value to a list element, you must use a valid index for an existing element, or an `IndexError` exception will occur. For example, look at the following code:

```
numbers = [1, 2, 3, 4, 5]   # Create a list with 5 elements.
numbers[5] = 99             # This raises an exception!
```

The `numbers` list that is created in the first statement has five elements, with the indexes 0 through 4. The second statement will raise an `IndexError` exception because the `numbers` list has no element at index 5.

If you want to use indexing expressions to fill a list with values, you have to create the list first, as shown here:

```
1  # Create a list with 5 elements.
2  numbers = [0] * 5
3
4  # Fill the list with the value 99.
5  index = 0
6  while index < len(numbers):
7      numbers[index] = 99
8      index += 1
```

The statement in line 2 creates a list with five elements, each element assigned the value 0. The loop in lines 6 through 8 then steps through the list elements, assigning 99 to each one.

Program 8-10 shows an example of how user input can be assigned to the elements of a list. This program gets sales amounts from the user and assigns them to a list.

Program 8-10 (sales_list.py)

```
# The NUM-DAYS constant holds the number of
# days that we will gather sales data for.
NUM-DAYS = 5

def main():
    # Create a list to hold the sales
    # for each day.
    sales = [0] * NUM_DAYS

    # Create a variable to hold an index.
    index = 0

    print 'Enter the sales for each day.'

    # Get the sales for each day.
    while index < NUM-DAYS:
        sales[index] = input('Day #' + str(index+1) + ': ')
        index += 1

```

(program continues)

Program 8-10 *(continued)*

```
20      # Display the values entered.
21      print 'Here are the values you entered:'
22      for value in sales:
23          print value
24
25  # Call the main function.
26  main()
```

Program Output (with input shown in bold)

```
Enter the sales for each day.
Day #1: 1000 [Enter]
Day #2: 2000 [Enter]
Day #3: 3000 [Enter]
Day #4: 4000 [Enter]
Day #5: 5000 [Enter]
Here are the values you entered:
1000
2000
3000
4000
5000
```

The statement in line 3 creates the variable `NUM_DAYS`, which is used as a constant for the number of days. The statement in line 8 creates a list with five elements, with each element assigned the value 0. Line 11 creates a variable named `index` and assigns the value 0 to it.

The loop in lines 16 through 18 iterates 5 times. The first time it iterates, `index` will reference the value 0, so the statement in line 17 assigns the user's input to `sales[0]`. The second time the loop iterates, `index` will reference the value 1, so the statement in line 17 assigns the user's input to `sales[1]`. This continues until input values have been assigned to all of the elements in the list.

List Methods

Lists have numerous methods that allow you to add elements, remove elements, change the ordering of elements, and so forth. We will look at a few of these methods: which are listed in Table 8-4.

The append Method

The `append` method is commonly used to add items to a list. The item that is passed as an argument is appended to the end of the list's existing elements. Program 8-11 shows an example,

---

[2] We do not cover all of the list methods in this book. For a description of all of the list methods, see the Python documentation at www.python.org.

Table 8-4 A few of the list methods

Method	Description
`append(item)`	Adds `item` to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts `item` into the list at the specified `index`. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of `item` from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

Program 8-11 (list_append.py)

```
# This program demonstrates how the append
# method can be used to add items to a list.

def main():
    # First, create an empty list.
    name-list = []

    # Create a variable to control the loop.
    again = 'Y'

    # Add some names to the list.
    while again.upper() == 'Y':
       # Get a name from the user.
       name = raw-input('Enter a name: ')

       # Append the name to the list.
       name_list.append(name)

       # Add another one?
       print 'Do you want to add another name?'
       again = raw-input('Y = yes, anything else = no: ')
       print

```

(program *continues*)

Program 8-11 *(continued)*

```
24     # Display the names that were entered.
25     print 'Here are the names you entered.'
26
27     for name in name-list:
28         print name
29
30 # Call the main function.
31 main()
```

Program Output (with input shown in bold)

```
Enter a name: Kathryn [Enter]
Do you want to add another name?
Y = yes, anything else = no: y [Enter]

Enter a name: Chris [Enter]
Do you want to add another name?
Y = yes, anything else = no: y [Enter]

Enter a name: Kenny [Enter]
Do you want to add another name?
Y = yes, anything else = no: y [Enter]

Enter a name: Renee [Enter]
Do you want to add another name?
Y = yes, anything else = no: n [Enter]

Here are the names you entered.
Kathryn
Chris
Kenny
Renee
```

Notice the statement in line 6:

```
name-list = []
```

This statement creates an empty list (a list with no elements) and assigns it to the `name_list` variable. Inside the loop, the `append` method is called to build the list. The first time the method is called, the argument passed to it will become element 0. The second time the method is called, the argument passed to it will become element 1. This continues until the user exits the loop.

The `index` Method

Earlier you saw how the `in` operator can be used to determine whether an item is in a list. Sometimes you need to know not only whether an item is in a list, but where it is located. The `index` method is useful in these cases. You pass an argument to the `index` method and it returns the index of the first element in the list containing that item. If the item is not found in the list, the method raises a `ValueError` exception. Program 5-12 demonstrates the `index` method.

Program 8-12 (index_list.py)

```
1   # This program demonstrates how  to get the
2   # index of an item in a list and then replace
3   # that item with a new item.
4
5   def main():
6       # Create a list with some items.
7       food = ['Pizza', 'Burgers', 'Chips']
8
9       # Display the list.
10      print 'Here are the items in the food list:'
11      print food
12
13      # Get the item to change.
14      item = raw_input('Which item should I change? ')
15
16      try:
17          # Get the item's index in the list.
18          item-index = food.index(item)
19
20          # Get the value to replace it with.
21          new-item = raw_input('Enter the new value: ')
22
23          # Replace the old item with the new item.
24          food[item_index] = new-item
25
26          # Display the list.
27          print 'Here is the revised list:'
28          print food
29      except ValueError:
30          print 'That item was not found in the list.'
31
32  # Call the main function.
33  main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I change? Burgers [Enter]
Enter the new value: Pickles [Enter]
Here is the revised list:
['Pizza', 'Pickles', 'Chips']
```

The elements of the `food` list are displayed in line 11, and in line 14 the user is asked which item he or she wants to change. Line 18 calls the `index` method to get the index of the item. Line 21 gets the new value from the user, and line 24 assigns the new value to the element holding the old value.

The insert Method

The `insert` method allows you to insert an item into a list at a specific position. You pass two arguments to the `insert` method: an index specifying where the item should be inserted and the item that you want to insert. Program 8-13 shows an example.

Program 8-13 (insert_list.py)

```
# This program demonstrates the insert method.

def main():
    # Create a list with some names.
    names = ['James', 'Kathryn', 'Bill']

    # Display the list.
    print 'The list before the insert:'
    print names

    # Insert a new name at element 0.
    names.insert(0, 'Joe')

    # Display the list again.
    print 'The list after the insert:'
    print names

# Call the main function.
main()
```

Program Output

```
The list before the insert:
['James', 'Kathryn', 'Bill']
The list after the insert:
['Joe', 'James', 'Kathryn', 'Bill']
```

The sort Method

The `sort` method rearranges the elements of a list so they appear in ascending order (from the lowest value to the highest value). Here is an example:

```
my-list= [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
print 'Original order:', my-list
my_list.sort()
print 'Sorted order:', my-list
```

When this code runs it will display the following:

```
Original order: [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
Sorted order: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here is another example:

```
my-list = ['beta', 'alpha', 'delta', 'gamma']
print 'Original order:', my-list
my_list.sort()
print 'Sorted order:', my-list
```

When this code runs it will display the following:

```
Original order: ['beta', 'alpha', 'delta', 'gamma']
Sorted order: ['alpha', 'beta', 'delta', 'gamma']
```

The `remove` Method

The **`remove`** method removes an item from the list. You pass an item to the method as an argument and the first element containing that item is removed. This reduces the size of the list by one element. All of the elements after the removed element are shifted one position toward the beginning of the list. A `ValueError` exception is raised if the item is not found in the list. Program 8-14 demonstrates the method.

Program 8-14 (remove-item.py)

```
# This program demonstrates how to use the remove
# method to remove an item from a list.

def main():
    # Create a list with some items.
    food = ['Pizza', 'Burgers', 'Chips']

    # Display the list.
    print 'Here are the items in the food list:'
    print food

    # Get the item to change.
    item = raw-input('Which item should I remove? ')

    try:
        # Remove the item.
        food.remove(item)

        # Display the list.
        print 'Here is the revised list:'
        print food

    except ValueError:
```

(program continues)

Program 8-14 *(continued)*

```
        print 'That item was not found in the list.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I remove? Burgers [Enter]
Here is the revised list:
['Pizza', 'Chips']
```

The reverse Method

The `reverse` method simply reverses the order of the items in the list. Here is an example:

```
my-list = [1, 2, 3, 4, 5]
print 'Original order:', my-list
my_list.reverse()
print 'Reversed:', my-list
```

This code will display the following:

```
Original order: [1, 2, 3, 4, 5]
Reversed: [5, 4, 3, 2, 1]
```

The `del` Statement

The `remove` method that you saw earlier removes a specific item from a list, if that item is in the list. Some situations might require that you remove an element from a specific index, regardless of the item that is stored at that index. This can be accomplished with the `del` statement. Here is an example of how to use the `del` statement:

```
my-list = [1, 2, 3, 4, 5]
print 'Before deletion:', my-list
del my-list[2]
print 'After deletion:', my-list
```

This code will display the following:

```
Before deletion: [1, 2, 3, 4, 5]
After deletion: [1, 2, 4, 5]
```

The `min` and `max` Functions

Python has two built-in functions named `min` and `max` that work with sequences. The `min` function accepts a sequence, such as a list or a string, as an argument and returns the item that has the lowest value in the sequence. Here is an example:

```
my-list = [5, 4, 3, 2, 50, 40, 30]
print 'The lowest value is', min(my_list)
```

This code will display the following:

```
The lowest value is 2
```

The max function accepts a sequence, such as a list or a string, as an argument and returns the item that has the highest value in the sequence. Here is an example:

```
my-list = [5, 4, 3, 2, 50, 40, 30]
print 'The highest value is', max(my_list)
```

This code will display the following:

```
The highest value is 50
```

Concatenating Lists

You can use the + operator to concatenate two lists. Here is an example:

```
listl = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list3 = listl + list2
```

After this code executes, `list3` will reference the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

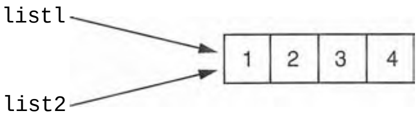
Copying Lists

Recall that in Python, assigning one variable to another variable simply makes both variables reference the same object in memory. For example, look at the following code:

```
# Create a list.
listl = [1, 2, 3, 4]
# Assign the list to the list2 variable.
list2 = listl
```

After this code executes, both variables `listl` and `list2` will reference the same list in memory. This is shown in Figure 8-8.

Figure 8-8 `listl` and `list2` reference the same list



Suppose you wish to make a copy of the list, so that `list1` and `list2` reference two separate but identical lists. One way to do this is with a loop that copies each element of the list. Here is an example:

```
# Create a list with values.
listl = [1, 2, 3, 4]
# Create an empty list.
```

```
list2 = []
# Copy the elements of listl to list2.
for item in listl:
    list2.append(item)
```

After this code executes, `list1` and `list2` will reference two separate but identical lists. A simpler and more elegant way to accomplish the same task is to use the concatenation operator, as shown here:

```
# Create a list with values.
listl = [1, 2, 3, 4]
# Create a copy of listl.
list2 = [] + listl
```

The last statement in this code concatenates an empty list with **`listl`** and assigns the resulting list to `list2`. As a result, `list1` and `list2` will reference two separate but identical lists.

Processing Lists

So far you've learned a wide variety of techniques for working with lists. Now we will look at a number of ways that programs can process the data held in a list. For example, the following In *the Spotlight* section shows how list elements can be used in calculations.

In the Spotlight:

Using List Elements in a Math Expression

Megan owns a small neighborhood coffee shop, and she has six employees who work as baristas (coffee bartenders). All of the employees have the same hourly pay rate. Megan has asked you to design a program that will allow her to enter the number of hours worked by each employee and then display the amounts of all the employees' gross pay. You determine that the program should perform the following steps:

1. For each employee: get the number of hours worked and store it in a list element.
2. For each list element: use the value stored in the element to calculate an employee's gross pay. Display the amount of the gross pay.

Program 8-15 shows the code for the program.

Program 8-15 (barista_pay.py)

```
# This program calculates the gross pay for
# each of Megan's baristas.

# NUM_EMPLOYEES is used as a constant for the
```

```
 5  # size of the list.
 6  NUM_EMPLOYEES = 6
 7
 8  def main():
 9      # Create a list to hold employee hours.
10      hours = [0] * NUM_EMPLOYEES
11
12      # Get each employee's hours worked.
13      for index in range(NUM_EMPLOYEES):
14          print 'Enter the hours worked by employee',
15          hours[index] = input(str(index + 1) + ': ')
16
17      # Get the hourly pay rate.
18      pay_rate = input('Enter the hourly pay rate: ')
19
20      # Display each employee's gross pay.
21      for index in range(NUM_EMPLOYEES):
22          gross_pay = hours[index] * pay_rate
23          print 'Gross pay for employee', index + 1, 'is',
24          print '$%.2f' % gross_pay
25
26  # Call the main function.
27  main()
```

Program Output (with input shown in bold)

```
Enter the hours worked by employee 1: 10 [Enter]
Enter the hours worked by employee 2: 20 [Enter]
Enter the hours worked by employee 3: 15 [Enter]
Enter the hours worked by employee 4: 40 [Enter]
Enter the hours worked by employee 5: 20 [Enter]
Enter the hours worked by employee 6: 18 [Enter]
Enter the hourly pay rate: 12.75 [Enter]
Gross pay for employee 1: $127.50
Gross pay for employee 2: $255.00
Gross pay for employee 3: $191.25
Gross pay for employee 4: $510.00
Gross pay for employee 5: $255.00
Gross pay for employee 6: $229.50
```

NOTE: Suppose Megan's business increases and she hires two additional baristas. This would require you to change the program so it processes eight employees instead of six. Because you used a constant for the list size, this is a simple modification—you just change the statement in line 6 to read:

```
NUM_EMPLOYEES = 8
```

(continued)

Because the `NUM_EMPLOYEES` constant is used in line 10 to create the list, the size of the `hours` list will automatically become eight. Also, because you used the `NUM_EMPLOYEES` constant to control the loop iterations in lines 13 and 21, the loops will automatically iterate eight times, once for each employee.

Imagine how much more difficult this modification would be if you had not used a constant to determine the list size. You would have to change each individual statement in the program that refers to the list size. Not only would this require more work, but it would open the possibility for errors. If you overlooked any one of the statements that refer to the list size, a bug would occur.

Totaling the Values in a List

Assuming a list contains numeric values, to calculate the total of those values you use a loop with an accumulator variable. The loop steps through the list, adding the value of each element to the accumulator. Program 8-16 demonstrates the algorithm with a list named `numbers`.

Program 8-16 (total_list.py)

```
# This program calculates the total of the values
# in a list.

def main():
    # Create a list.
    numbers = [2, 4, 6, 8, 10]

    # Create a variable to use as an accumulator.
    total = 0

    # Calculate the total of the list elements.
    for value in numbers:
        total += value

    # Display the total of the list elements.
    print 'The total of the elements is', total

# Call the main function.
main()
```

Program Output

```
The total of the elements is 30
```

Averaging the Values In a List

The first step in calculating the average of the values in a list is to get the total of the values. You saw how to do that with a loop in the preceding section. The second step is to divide the total by the number of elements in the list. Program 8-17 demonstrates the algorithm.

Program 8-17 (average_list.py)

```
# This program calculates the average of the values
# in a list.

def main():
    # Create a list.
    scores = [2.5, 8.3, 6.5, 4.0, 5.21

    # Create a variable to use as an accumulator.
    total = 0.0

    # Calculate the total of the list elements.
    for value in scores:
        total += value

    # Calculate the average of the elements.
    average = total / len(scores)

    # Display the total of the list elements.
    print 'The average of the elements is', average

# Call the main function.
main()
```

Program Output

```
The average of the elements is 5.3
```

Passing a List as an Argument to a Function

'Recall from Chapter **3** that as a program grows larger and more complex, it should be broken down into functions that each performs a specific task. This makes the program easier to understand and to maintain.

You can easily pass a list as an argument to a function. This gives you the ability to put many of the operations that you perform on a list in their own functions. When you need to call these functions, you can pass the list as an argument.

Program 8-18 shows an example of a program that uses such a function. The function in this program accepts a list as an argument and returns the total of the list's elements.

Program 8-18 (total_function.py)

```
# This program uses a function to calculate the
# total of the values in a list.

def main():
    # Create a list.
    numbers = [2, 4, 6, 8, 10]

    # Display the total of the list elements.
    print 'The total is', get_total(numbers)

# The get-total function accepts a list as an
# argument returns the total of the values in
# the list.
def get_total(value_list):
    # Create a variable to use as an accumulator.
    total = 0

    # Calculate the total of the list elements.
    for num in value-list:
        total += num

    # Return the total
    return total

# Call the main function.
main()
```

Program Output

```
The total is 30
```

Returning a List From a Function

A function can return a reference to a list. This gives you the ability to write a function that creates a list and adds elements to it, and then returns a reference to the list so other parts of the program can work with it. The code in Program 8-19 shows an example. It uses a function named `get_values` that gets a series of values from the user, stores them in a list, and then returns a reference to the list.

Program 8-19 (return_list.py)

```
# This program uses a function to create a list.
# The function returns a reference to the list.

```

```
def main():
    # Get a list with values stored in it.
    numbers = get-values()

    # Display the values in the list.
    print 'The numbers in the list are:'
    print numbers

# The get-values function gets a series of numbers
# from the user and stores them in a list. The
# function returns a reference to the list.
def get-values():
    # Create an empty list.
    values = []

    # Create a variable to control the loop.
    again = 'Y'

    # Get values from the user and add them to
    # the list.
    while again.upper() == 'Y':
        # Get a number and add it to the list.
        num = input('Enter a number: ')
        values.append(num)

        # Want to do this again?
        print 'Do you want to add another number?'
        again = raw-input('Y = yes, anything else = no: ')
        print

    # Return the list.
    return values

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 1 [Enter]
Do you want to add another number?
Y = yes, anything else = no: y [Enter]

Enter a number: 2 [Enter]
Do you want to add another number?
Y = yes, anything else = no: y [Enter]

Enter a number: 3 [Enter]
Do you want to add another number?
Y = yes, anything else = no: y [Enter]
```

(program output continues)

Program Output *(continued)*

```
Enter a number: 4 [Enter]
Do you want to add another number?
Y = yes, anything else = no: y [Enter]

Enter a number: 5 [Enter]
Do you want to add another number?
Y = yes, anything else = no: n [Enter]

The numbers in the list are:
[1, 2, 3, 4, 5]
```

In the Spotlight:

Processing a List

Dr. LaClaire gives a series of exams during the semester in her chemistry class. At the end of the semester she drops each student's lowest test score before averaging the scores. She has asked you to design a program that will read a student's test scores as input, and calculate the average with the lowest score dropped. Here is the algorithm that you developed:

```
Get the student's test scores.
Calculate the total of the scores.
Find the lowest score.
Subtract the lowest score from the total. This gives the adjusted total.
Divide the adjusted total by 1 less than the number of test scores. This is the average.
Display the average.
```

Program 8-20 shows the code for the program, which is divided into three functions. Rather than presenting the entire program at once, let's first examine the `main` function and then each additional function separately. Here is the `main` function:

Program 8-20 drop_lowest_score.py: `main` function

```
# This program gets a series of test scores and
# calculates the average of the scores with the
# lowest score dropped.

def main():
    # Get the test scores from the user.
    scores = get_scores()

    # Get the total of the test scores.
    total = get_total(scores)

    # Get the lowest test score.
```

```
13      lowest = min(scores)
14
15      # Subtract the lowest score from the total.
16      total -= lowest
17
18      # Calculate the average. Note the following:
19      # (1) The float function is used to make sure
20      # total is a float, thus preventing integer
21      # division.
22      # (2) We divide by 1 less than the number of
23      # scores because the lowest score was dropped.
24      average = float(total) / (len(scores) - 1)
25
26      # Display the average.
27      print 'The average, with the lowest score dropped'
28      print 'is', average
29
```

Line 7 calls the `get_scores` function. The function gets the test scores from the user and returns a reference to a list containing those scores. The list is assigned to the scores variable.

Line 10 calls the `get_total` function, passing the `scores` list as an argument. The function returns the total of the values in the list. This value is assigned to the `total` variable.

Line 13 calls the built-in `min` function, passing the `scores` list as an argument. The function returns the lowest value in the list. This value is assigned to the `lowest` variable.

Line 16 subtracts the lowest test score from the `total` variable. Then, line 24 calculates the average by dividing `total` by `len(scores) – 1`. (The program divides by `len(scores) – 1` because the lowest test score was dropped.) Lines 27 and 28 display the average.

Next is the `get_scores` function.

Program 8-20 drop_lowest_score.py: `get_scores` function

```
30  # The get_scores function gets a series of test
31  # scores from the user and stores them in a list.
32  # A reference to the list is returned.
33  def get_scores():
34      # Create an empty list.
35      test_scores = []
36
37      # Create a variable to control the loop.
38      again = 'Y'
39
40      # Get the scores from the user and add them to
41      # the list.
```

(program continues)

Program 8-19 *(continued)*

```
42      while again.upper() == 'Y':
43          # Get a score and add it to the list.
44          value = input('Enter a test score: ')
45          test_scores.append(value)
46
47          # Want to do this again?
48          print 'Do you want to add another score?'
49          again = raw_input('Y = yes, anything else = no: ')
50          print
51
52      # Return the list.
53      return test_scores
54
```

The `get_scores` function prompts the user to enter a series of test scores. As each score is entered it is appended to a list. The list is returned in line 53. Next is the `get_total` function.

Program 8-20 drop_lowest_score.py: `get_total` function

```
55  # The get_total function accepts a list as an
56  # argument returns the total of the values in
57  # the list.
58  def get_total(value_list):
59      # Create a variable to use as an accumulator.
60      total = 0.0
61
62      # Calculate the total of the list elements.
63      for num in value_list:
64          total += num
65
66      # Return the total.
67      return total
68
69  # Call the main function.
70  main()
```

This function accepts a list as an argument. It uses an accumulator and a loop to calculate the total of the values in the list. Line 67 returns the total.

Program Output (with input shown in bold)

```
Enter a test score: 92 [Enter]
Do you want to add another score?
Y = yes, anything else = no: y [Enter]

Enter a test score: 67 [Enter]
Do you want to add another score?
Y = yes, anything else = no: y [Enter]
```

```
Enter a test score: 75 [Enter]
Do you want to add another score?
Y = yes, anything else = no: y [Enter]

Enter a test score: 88 [Enter]
Do you want to add another score?
Y = yes, anything else = no: n [Enter]

The average, with the lowest score dropped
is 85.0
```

Working with Lists and Files

Some tasks may require you to save the contents of a list to a file so the data can be used at a later time. Likewise, some situations may require you to read the data from a file into a list. For example, suppose you have a file that contains a set of values that appear in random order and you want to sort the values. One technique for sorting the values in the file would be to read them into a list, call the list's `sort` method, and then write the values in the list back to the file.

Saving the contents of a list to a file is a straightforward procedure. In fact, Python file objects have a method named `writelines` that writes an entire list to a file. A drawback to the `writelines` method, however, is that it does not automatically write a newline (`'\n'`) at the end of each item. Consequently, each item is written to one long line in the file. Program 8-21 demonstrates the method.

Program 8-21 (writelines.py)

```
# This program uses the writelines method to save
# a list of strings to a file.

def main():
    # Create a list of strings.
    cities = ['New York', 'Boston', 'Atlanta', 'Dallas']

    # Open a file for writing.
    outfile = open('cities.txt', 'w')

    # Write the list to the file.
    outfile.writelines(cities)

    # Close the file.
    outfile.close()

# Call the main function.
main()
```

After this program executes, the `cities.txt` file will contain the following line:

```
New YorkBostonAtlantaDallas
```

An alternative approach is to use the `for` loop to iterate through the list, writing each element with a terminating newline character. Program 8-22 shows an example.

Program 8-22 (write_list.py)

```
# This program saves a list of strings to a file.

def main():
    # Create a list of strings.
    cities = ['New York', 'Boston', 'Atlanta', 'Dallas']

    # Open a file for writing.
    outfile = open('cities.txt', 'w')

    # Write the list to the file.
    for item in cities:
        outfile.write(item + '\n')

    # Close the file.
    outfile.close()

# Call the main function.
main()
```

After this program executes, the `cities.txt` file will contain the following lines:

```
New York
Boston
Atlanta
Dallas
```

File objects in Python have a method named `readlines` that returns a file's contents as a list of strings. Each line in the file will be an item in the list. The items in the list will include their terminating newline character, which in many cases you will want to strip. Program 8-23 shows an example. The statement in line 8 reads the files contents into a list, and the loop in lines 15 through 17 steps through the list, stripping the `'\n'` character from each element.

Program 8-23 (read_list.py)

```
1  # This program reads a file's contents into a list.
2
3  def main():
4      # Open a file for reading.
```

```
    infile = open('cities.txt', 'r')

    # Read the contents of the file into a list.
    cities = infile.readlines()

    # Close the file.
    infile.close()

    # Strip the \n from each element.
    index = 0
    while index < len(cities):
        cities[index] = cities[index].rstrip('\n')
        index += 1

    # Print the contents of the list.
    print cities

# Call the main function.
main()
```

Program Output

```
['New York', 'Boston', 'Atlanta', 'Dallas']
```

Program 8-24 shows another example of how a list can be written to a file. In this example, a list of numbers is written. Notice that in line 12, each item is converted to a string with the `str` function, and then a `'\n'` is concatenated to it.

Program 8-24 (write_number_list.py)

```
 1  # This program saves a list of numbers to a file.
 2
 3  def main():
 4      # Create a list of numbers.
 5      numbers = [1, 2, 3, 4, 5, 6, 7]
 6
 7      # Open a file for writing.
 8      outfile = open('numberlist.txt', 'w')
 9
10      # write the list to the file.
11      for item in numbers:
12          outfile.write(str(item) + '\n')
13
14      # Close the file.
15      outfile.close()
16
17  # Call the main function.
18  main()
```

When you read numbers from a file into a list, the numbers will have to be converted from strings to a numeric type. Program 8-25 shows an example.

Program 8-25 (read–number–list.py)

```
# This program reads numbers from a file into a list.

def main():
    # Open a file for reading.
    infile = open('numberlist.txt', 'r')

    # Read the contents of the file into a list.
    numbers = infile.readlines()

    # Close the file.
    infile.close()

    # Convert each element to an int.
    index = 0
    while index < len(numbers):
        numbers[index] = int(numbers[index])
        index += 1

    # Print the contents of the list.
    print numbers

# Call the main function.
main()
```

Program Output

```
[1, 2, 3, 4, 5, 6, 7]
```

Splitting a String

Strings in Python have a method named split that returns a list of the words in the string. Program 8-26 shows an example.

Program 8-26 (string_split.py)

```
# This program demonstrates the split method.

def main():
    # Create a string with multiple words.
    my-string = 'One two three four'

    # Split the string.
    word-list = my_string.split()

    # Print the list of words.
    print word-list

# Call the main function.
main()
```

Program Output

```
['One', 'two', 'three', 'four']
```

By default, the `split` method uses spaces as separators (that is, it returns a list of the words in the string that are separated by spaces). You can specify a different separator by passing it as an argument to the `split` method. For example, suppose a string contains a date, as shown here:

```
date-string = '3/26/2008'
```

If you want break out the month, day, and year as items in a list, you can call the `split` method using the `'/'` character as a separator, as shown here:

```
date-list = date_string.split('/')
```

After this statement executes, the `date_list` variable will reference this list:

```
['3', '26', '2008']
```

Program 8-27 demonstrates this.

Program 8-27 (split_date.py)

```
# This program calls the split method, using the
# '/' character as a separator.

def main():
    # Create a string with a date.
    date-string = '11/26/2008'

    # Split the date.
    date-list = date_string.split('/')

    # Display each piece of the date.
    print 'Month:', date_list[0]
    print 'Day:', date_list[1]
    print 'Year:', date_list[2]

```

(program continues)

Program 8-27 *(continued)*

```
# Call the main function.
main()
```

Program Output

```
Month: 11
Day: 26
Year: 2008
```

Checkpoint

8.18 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
numbers[2] = 99
print numbers
```

8.19 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my-list = numbers[1:3]
print my-list
```

8.20 How do you find the number of elements in a list?

8.21 What is the difference between calling a list's `remove` method and using the `del` statement to remove an element?

8.22 How do you find the lowest and highest values in a list?

8.23 Assume the following statement appears in a program:

```
names = []
```

Which of the following statements would you use to add the string `'Wendy'` to the list at index 0? Why would you select this statement instead of the other?

a. `names[0] = 'Wendy'`
b. `names.append('Wendy')`

8.24 Describe the following list methods:

a. `index`
b. `insert`
c. `sort`
d. `reverse`

8.25 Assume the following statement appears in a program:

```
days = 'Monday Tuesday Wednesday'
```

Write a statement that splits the string, creating the following list:

```
['Monday', 'Tuesday', 'Wednesday']
```

8.26 Briefly describe how you calculate the total of the values in a list.

8.27 Briefly describe how you get the average of the values in a list.

Review Questions

Multiple Choice

1. This term refers to an individual item in a list.
   a. element
   b. bin
   c. cubby hole
   d. slot
2. This is a number that identifies a character in a string, or an item in a list.
   a. element
   b. index
   c. bookmark
   d. identifier
3. This is the first index in a string or a list.
   a. −1
   b. 1
   c. 0
   d. The size of the list minus one
4. This is the last index in a string or a list.
   a. 1
   b. 99
   c. 0
   d. The size of the string or list minus one
5. This will happen if you try to use an index that is out of range for a list.
   a. a `ValueError` exception will occur
   b. an `IndexError` exception will occur
   c. The string or list will be erased and the program will continue to run.
   d. Nothing—the invalid index will be ignored
6. This function returns the length of a string or a list.
   a. `length`
   b. `size`
   c. `len`
   d. `lengthof`
7. This string method returns a copy of the string with all leading whitespace characters removed.
   a. `lstrip`
   b. `rstrip`
   c. `remove`
   d. `strip_leading`
8. This string method returns the lowest index in the string where a specified substring is found.
   a. `first_index_of`
   b. `locate`
   c. `find`
   d. `index_of`

9. This list method adds an item to the end of an existing list.
    a. `add`
    b. `add_to`
    c. `increase`
    d. `append`
10. This removes an item at a specific index in a list.
    a. The `remove` method
    b. The `delete` method
    c. The `del` statement
    d. The `kill` method
11. Assume the following statement appears in a program:

    `mylist = []`

    Which of the following statements would you use to add the string `'Labrador'` to the list at index 0?
    a. `mylist[0] = 'Labrador'`
    b. `mylist.insert(0, 'Labrador')`
    c. `mylist.append('Labrador')`
    d. `mylist.insert('Labrador', 0)`
12. If you call the `index` method to locate an item in a list and the item is not found, this happens.
    a. A `ValueError` exception is raised
    b. An `InvalidIndex` exception is raised
    c. The method returns −1
    d. Nothing happens. The program continues running at the next statement.
13. This built-in function returns the highest value in a list.
    a. `highest`
    b. `max`
    c. `greatest`
    d. `best_of`
14. This file object method returns a list containing the file's contents.
    a. `to_list`
    b. `getlist`
    c. `readline`
    d. `readlines`
15. Assume the following statement appears in a program:

    `mylist = 'four score and seven'`

    Which of the following expressions would you use to get the following list?

    `['four', 'score', 'and', 'seven']`

    a. `mylist.split()`
    b. `mylist.list()`
    c. `mylist.word_list()`
    d. `mylist.getlist()`

True or False

1. Strings in Python are immutable.
2. Lists in Python are immutable.
3. The `del` statement deletes an item at a specified index in a list.
4. Assume `list1` references a list. After the following statement executes, `list1` and `list2` will reference two identical but separate lists in memory:

   ```
   list2 = list1
   ```

5. A file object's `writelines` method automatically writes a newline (`'\n'`) after writing each list item to the file.

Short Answer

1. Look at the following statement:

   ```
   numbers = [10, 20, 30, 40, 50]
   ```

   a. How many elements does the list have?
   b. What is the index of the first element in the list?
   c. What is the index of the last element in the list?

3. Look at the following statement:

   ```
   numbers = [1, 2, 3]
   ```

   a. What value is stored in `numbers[2]`?
   b. What value is stored in `numbers[0]`?
   c. What value is stored in `numbers[-1]`?

4. What will the following code display?

   ```
   mystring = 'abcdefg'
   print mystring[2:5]
   ```

5. What does the following code display?

   ```
   numbers = [1, 2, 3, 4, 5, 6, 7]
   print numbers[4:6]
   ```

6. What does the following code display?

   ```
   name = 'joe'
   print name.lower()
   print name.upper()
   print name
   ```

Algorithm Workbench

1. Assume `choice` references a string. The following `if` statement determines whether `choice` is equal to 'Y' or 'y':

   ```
   if choice == 'Y' or choice == 'y':
   ```

   Rewrite this statement so it only makes one comparison and does not use the `or` operator. (*Hint: use either the* `upper` *or* `lower` *methods.*)

2. Write a loop that counts the number of space characters that appear in the string referenced by `mystring`.
3. Write a loop that counts the number of digits that appear in the string referenced by `mystring`.

4. Write a loop that counts the number of lowercase characters that appear in the string referenced by `mystring`.
5. Write a function that accepts a string as an argument and returns true if the argument ends with the substring `'.com'`. Otherwise, the function should return false.
6. Write code that makes a copy of a string with all occurrences of the lowercase letter `'t'` converted to uppercase.
7. Write a function that accepts a string as an argument and displays the string backwards.
8. Assume `mystring` references a string. Write a `print` statement that uses a slicing expression to display the first 3 characters in the string.
9. Assume `mystring` references a string. Write a `print` statement that uses a slicing expression to display the last 3 characters in the string.
10. Write a statement that creates a list with the following strings: `'Einstein'`, `'Newton'`, `'Copernicus'`, and `'Kepler'`.
11. Assume `names` references a list. Write a `for` loop that displays each element of the list.
12. Assume the lists `numbers1` has 100 elements and `numbers2` is an empty list. Write code that copies the values in `numbers1` to `numbers2`.
13. Draw a flowchart showing the general logic for totaling the values in a list.
14. Write a function that accepts a list as an argument (assume the list contains integers) and returns the total of the values in the list.
15. Look at the following statement:

    ```
    mystring = 'cookies>milk>fudge>cake>ice cream'
    ```

    Write a statement that splits this string, creating the following list:

    ```
    ['cookies', 'milk', 'fudge', 'cake', 'ice cream']
    ```

Programming Exercises

1. Initials

Write a program that gets a string containing a person's first, middle, and last names, and then display their first, middle, and last initials. For example, if the user enters John William Smith the program should display J. W. S.

2. Sum of Digits in a String

Write a program that asks the user to enter a series of single-digit numbers with nothing separating them. The program should display the sum of all the single digit numbers in the string. For example, if the user enters 2514, the method should return 12, which is the sum of 2, 5, 1, and 4.

3. Date Printer

Write a program that reads a string from the user containing a date in the form mm/dd/yyyy. It should print the date in the form March 12, 2009.

4. Morse Code Converter

Morse code is a code where each letter of the English alphabet, each digit, and various punctuation characters are represented by a series of dots and dashes. Table 8-5 shows part of the code.

Write a program that asks the user to enter a string, and then converts that string to Morse code.

Table 8-5 Morse code

Character	Code	Character	Code	Character	Code	Character	Code
space	*space*	6	–....	G	––.	Q	––.–
comma	––..––	7	––...	H	....	R	.–.
period	.–.–.–	8	–––..	I	..	S	...
question mark	..––..	9	––––.	J	.–––	T	–
0	–––––	A	.–	K	–.–	U	..–
1	.––––	B	–...	L	.–..	V	...–
2	..–––	C	–.–.	M	––	W	.––
3	...––	D	–..	N	–.	X	–..–
4	....–	E	.	O	–––	Y	–.–
5	.....	F	..–.	P	.––.	Z	––..

5. Alphabetic Telephone Number Translator

Many companies use telephone numbers like 555-GET-FOOD so the number is easier for their customers to remember. On a standard telephone, the alphabetic letters are mapped to numbers in the following fashion:

A, B, and C = 2
D, E, and F = 3
G, H, and I = 4
J, K, and L = 5
M, N, and O = 6
P, Q, R, and S = 7
T, U, and V = 8
W, X, Y, and Z = 9

Write a program that asks the user to enter a 10-character telephone number in the format XXX-XXX-XXXX. The application should display the telephone number with any alphabetic characters that appeared in the original translated to their numeric equivalent. For example, if the user enters 555-GET-FOOD the application should display 555-438-3663.

6. Average Number of Words

On the student CD you will find a file named `text.txt`. The text that is in the file is stored as one sentence per line. Write a program that reads the file's contents and calculates the average number of words per sentence.

(If a CD did not come with your book, or you cannot locate your CD, you can visit www.aw.com/cssupport to access this file.)

7. Character Analysis

On the student CD you will find a file named `text.txt`. Write a program that reads the file's contents and determines the following:

- The number of uppercase letters in the file
- The number of lowercase letters in the file
- The number of digits in the file
- The number of whitespace characters in the file

(If a CD did not come with your book, or you cannot locate your CD, you can visit www.aw.com/cssupport to access this file.)

8. Total Sales

Design a program that asks the user to enter a store's sales for each day of the week. The amounts should be stored in a list. Use a loop to calculate the total sales for the week and display the result.

9. Lottery Number Generator

Design a program that generates a seven-digit lottery number. The program should generate seven random numbers, each in the range of 0 through 9, and assign each number to a list element. (Random numbers were discussed in Chapter 6.) Then write another loop that displays the contents of the list.

10. Rainfall Statistics

Design a program that lets the user enter the total rainfall for each of 12 months into a list. The program should calculate and display the total rainfall for the year, the average monthly rainfall, and the months with the highest and lowest amounts.

11. Number Analysis Program

Design a program that asks the user to enter a series of 20 numbers. The program should store the numbers in a list and then display the following data:

- The lowest number in the list
- The highest number in the list
- The total of the numbers in the list
- The average of the numbers in the list

12. Charge Account Validation

On the student CD you will find a file named `charge_accounts.txt`. This file has a list of a company's valid charge account numbers. Each account number is a seven-digit number, such as `5658845`.

Write a program that reads the contents of the file into a list. The program should then ask the user to enter a charge account number. The program should determine whether the number is valid by searching for it in the list. If the number is in the list, the program should display a message indicating the number is valid. If the number is not in the list, the program should display a message indicating the number is invalid.

(If a CD did not come with your book, or you cannot locate your CD, you can visit www.aw.com/cssupport to access this file.)

CHAPTER 9

Classes and Object-Oriented Programming

TOPICS

9.1 Procedural and Object-Oriented Programming
9.2 Classes
9.3 Working with Instances
9.4 Techniques for Designing Classes

9.1 Procedural and Object-Oriented Programming

CONCEPT: Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered on objects. Objects are created from abstract data types that encapsulate data and functions together.

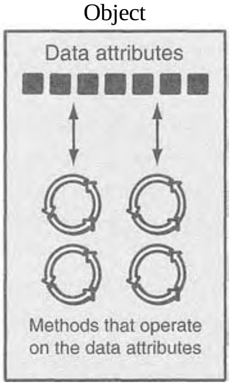
There are primarily two methods of programming in use today: procedural and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. You can think of a procedure simply as a function that performs a specific task such as gathering input from the user, performing calculations, reading or writing files, displaying output, and so on. The programs that you have written so far have been procedural in nature.

Typically, procedures operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another. As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data can lead to problems, however, as the program becomes larger and more complex.

For example, suppose you are part of a programming team that has written an extensive customer database program. The program was initially designed so that a customer's name, address, and phone number were referenced by three variables. Your job was to design several functions that accept those three variables as arguments and perform operations on them. The software has been operating successfully for some time, but your team has been asked to update it by adding several new features. During the revision process, the senior programmer informs you that the customer's name, address, and phone number will no longer be stored in variables. Instead, they will be stored in a list. This means that you will have to modify all of the functions that you have designed so that they accept and work with a list instead of the three variables. Malting these extensive modifications not only is a great deal of work, but also opens the opportunity for errors to appear in your code.

Whereas procedural programming is centered on creating procedures (functions), *object-oriented programming* (OOP)is centered on creating objects. An object is a software entity that contains both data and procedures. The data contained in an object is known as the object's data attributes. An object's data attributes are simply variables that reference data. The procedures that an object performs are known as methods. An object's methods are functions that perform operations on the object's data attributes. The object is, conceptually, a self-contained unit that consists of data attributes and methods that operate on the data attributes. This is illustrated in Figure 9-1.

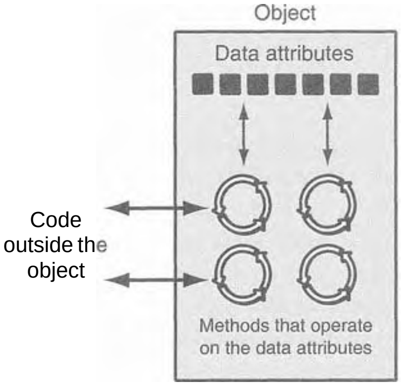
Figure 9-1 *An object contains data attributes and methods*



OOP addresses the problem of code and data separation through encapsulation and data hiding. Encapsulation refers to the combining of data and code into a single object. Data hiding refers to an object's ability to hide its data attributes from code that is outside the object. Only the object's methods may directly access and make changes to the object's data attributes.

An object typically hides its data, but allows outside code to access its methods. As shown in Figure 9-2, the object's methods provide programming statements outside the object with indirect access to the object's data attributes.

Figure 9-2 Code outside the object interacts with the object's methods



When an object's data attributes are hidden from outside code, and access to the data attributes is restricted to the object's methods, the data attributes are protected from accidental corruption. In addition, the code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's methods. When a programmer changes the structure of an object's internal data attributes, he or she also modifies the object's methods so that they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

Object Reusability

In addition to solving the problems of code and data separation, the use of OOP has also been encouraged by the trend of object reusability. An object is not a stand-alone program, but is used by programs that need its services. For example, Sharon is a programmer who has developed a set of objects for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her objects are coded to perform all of the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's objects to perform the 3D rendering (for a small fee, of course!).

An Everyday Example of an Object

Imagine that your alarm clock is actually a software object. If it were, it would have the following data attributes:

- `current_second` (a value in the range of 0–59)
- `current_minute` (a value in the range of 0–59)
- `current_hour` (a value in the range of 1–12)
- `alarm_time` (a valid hour and minute)
- `alarm_is_set` (True or False)

As you can see, the data attributes are merely values that define the *state* that the alarm clock is currently in. You, the user of the alarm clock object, cannot directly manipulate these data attributes because they are *private*. To change a data attribute's value, you must use one of the object's methods. The following are some of the alarm clock object's methods:

- `set_time`
- `set_alarm_time`
- `set_alarm_on`
- `set_alarm_off`

Each method manipulates one or more of the data attributes. For example, the `set_time` method allows you to set the alarm clock's time. You activate the method by pressing a button on top of the clock. By using another button, you can activate the `set_alarm_time` method.

In addition, another button allows you to execute the `set_alarm_on` and `set_alarm_off` methods. Notice that all of these methods can be activated by you, who are outside the alarm clock. Methods that can be accessed by entities outside the object are known as *public methods*.

The alarm clock also has *private methods*, which are part of the object's private, internal workings. External entities (such as you, the user of the alarm clock) do not have direct access to the alarm clock's private methods. The object is designed to execute these methods automatically and hide the details from you. The following are the alarm clock object's private methods:

- `increment_current_second`
- `increment_current_minute`
- `increment_current_hour`
- `sound_alarm`

Every second the `increment_current_second` method executes. This changes the value of the `current_second` data attribute. If the `current_second` data attribute is set to 59 when this method executes, the method is programmed to reset `current_second` to 0, and then cause the `increment_current_minute` method to execute. This method adds 1 to the `current_minute` data attribute, unless it is set to 59. In that case, it resets `current_minute` to 0 and causes the `increment_current_hour` method to execute. The `increment_current_minute` method compares the new time to the `alarm_time`. If the two times match and the alarm is turned on, the `sound_alarm` method is executed.

Checkpoint

9.1 What is an object?

9.2 What is encapsulation?

9.3 Why is an object's internal data usually hidden from outside code?

9.4 What are public methods? What are private methods?

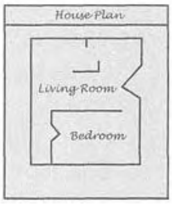
9.2 Classes

CONCEPT: **A** class is code that specifies the data attributes and methods for a particular type of object.

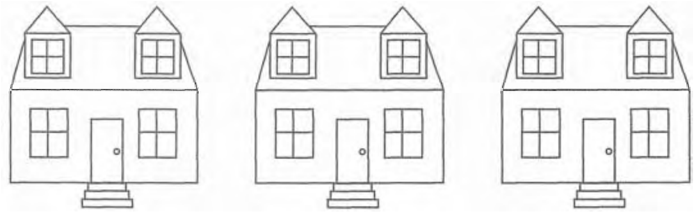
Now, let's discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the data attributes and methods that are necessary, and then creates a *class*. A class is code that specifies the data attributes and methods of a particular type of object. Think of a class as a "blueprint" that objects may be created from. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an *instance* of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 9-3.

Figure 9-3 A blueprint and houses built from the blueprint

Blueprint that describes a house



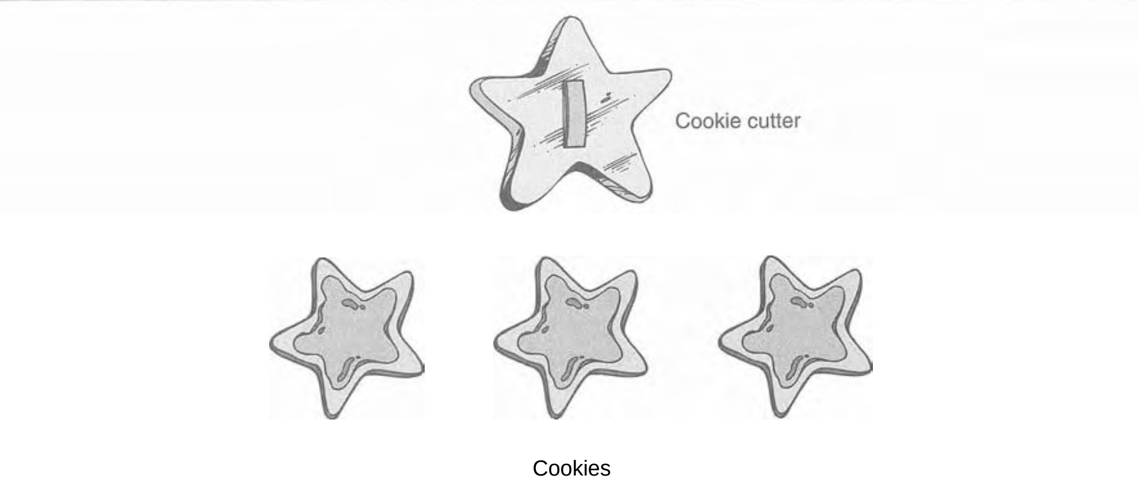
Instances of the house described by the blueprint



Another way of thinking about the difference between a class and an object is to think of the difference between a cookie cutter and a cookie. While a cookie cutter itself is not a cookie, it describes a cookie. The cookie cutter can be used to make several cookies, as shown in Figure 9-4. Think of a class as a cookie cutter and the objects created from the class as cookies.

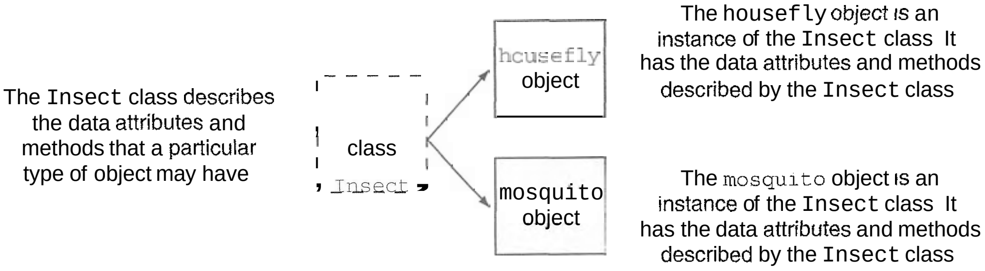
So, a class is a description of an object's characteristics. When the program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

Figure 9-4 The cookie cutter metaphor



For example, Jessica is an entomologist (someone who studies insects) and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies characteristics that are common to all *types* of insects. The `Insect` class is a specification that objects may be created from. Next, she writes programming statements that create an object named `housefly`, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a housefly. It has the data attributes and methods specified by the `Insect` class. Then she writes programming statements that create an object named `mosquito`. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory, and stores data about a mosquito. Although the `housefly` and `mosquito` objects are separate entities in the computer's memory, they were both created from the `Insect` class. This means that each of the objects has the data attributes and methods described by the `Insect` class. This is illustrated in Figure 9-5.

Figure 9-5 The `housefly` and `mosquito` objects are instances of the **`Insect`** class



The `Insect` class describes the data attributes and methods that a particular type of object may have

The `housefly` object is an instance of the `Insect` class It has the data attributes and methods described by the `Insect` class

The `mosquito` object is an instance of the `Insect` class It has the data attributes and methods described by the `Insect` class

Class Definitions

To create a class, you write a *class* definition. A class definition is a set of statements that define a class's methods and data attributes. Let's look at a simple example. Suppose we are writing a program to simulate the tossing of a coin. In the program we need to repeatedly toss the coin and each time determine whether it landed heads up or tails up. Taking an object-oriented approach, we will write a class named `Coin` that can perform the behaviors of the coin.

Program 9-1 shows the class definition, which we will explain shortly. Note that this is not a complete program. We will add to it as we go along.

Program 9-1 (Coin class, not a complete program)

```
1   import random
2
3   # The Coin class simulates a coin that can
4   # be flipped.
5
6   class Coin:
7
8       # The __init__ method initializes the
9       # sideup data attribute with 'Heads'.
10
11      def __init__(self):
12          self.sideup = 'Heads'
13
14      # The toss method generates a random number
15      # in the range of 0 through 1. If the number
16      # is 0, then sideup is set to 'Heads'.
17      # Otherwise, sideup is set to 'Tails'.
18
19      def toss(self):
20          if random.randint(0, 1) == 0:
21              self.sideup = 'Heads'
22          else:
23              self.sideup = 'Tails'
24
25      # The get-sideup method returns the value
26      # referenced by sideup.
27
28      def get_sideup(self):
29          return self.sideup
```

In line 1 we import the `random` module. This is necessary because we use the `randint` function to generate a random number. Line 6 is the beginning of the class definition. It begins with the keyword `class`, followed by the class name, which is `coin`, followed by a colon.

The same rules that apply to variable names also apply to class names. However, notice that we started the class name, `Coin`, with an uppercase letter. This is not a requirement, but it is a widely used convention among programmers. This helps to easily distinguish class names from variable names when reading code.

The `coin` class has three methods:

- The `__init__` method appears in lines 11 through 12.
- The `toss` method appears in lines 19 through 23.
- The `get_sideup` method appears in lines 28 through 29.

Except for the fact that they appear inside a class, notice that these method definitions look like any other function definition in Python. They start with a header line, which is followed by an indented block of statements.

Take a closer look at the header for each of the method definitions (lines 11, 19, and 28) and notice that each method has a parameter variable named `self`:

Line 11: `def __init__(self):`
Line 19: `def toss(self):`
Line 28: `def get_sideup(self):`

The `self` parameter[1] is required in every method of a class. Recall from our earlier discussion on object-oriented programming that a method operates on a specific object's data attributes. When a method executes, it must have a way of knowing which object's data attributes it is supposed to operate on. That's where the `self` parameter comes in. When a method is called, Python makes the `self` parameter reference the specific object that the method is supposed to operate on.

Let's look at each of the methods. The first method, which is named `__init__`, is defined in lines 11 through 12:

```
def __init__(self):
    self.sideup = 'Heads'
```

Most Python classes have a special method named `__init__`, which is automatically executed when an instance of the class is created in memory. The `__init__` method is commonly known as an *initializer method* because it initializes the object's data attributes. (The name of the method starts with two underscore characters, followed by the word `init`, followed by two more underscore characters.)

Immediately after an object is created in memory, the `__init__` method executes, and the `self` parameter is automatically assigned the object that was just created. Inside the method, the statement in line 12 executes:

```
self.sideup = 'Heads'
```

This statement assigns the string `'Heads'` to the `sideup` data attribute belonging to the object that was just created. As a result of this `__init__` method, each object that we create from the `Coin` class will initially have'a `sideup` attribute that is set to `'Heads'`.

NOTE: The `__init__` method is usually the first method inside a class definition.

---

[1] The parameter must be present in a method. You are not required to name it `self`, but this is strongly recommended to conform with standard practice.

The `toss` method appears in lines 19 through 23:

```
def toss(self):
    if random.randint(0, 1) == 0:
    self.sideup = 'Heads'
    else:
    self.sideup = 'Tails'
```

This method also has the required `self` parameter variable. When the `toss` method is called, `self` will automatically reference the object that the method is to operate on.

The `toss` method simulates the tossing of the coin. When the method is called, the `if` statement in line 20 calls the `random.randint` function to get a random integer in the range of 0 through 1. If the number is 0, then the statement in line 21 assigns `'Heads'` to `self.sideup`. Otherwise, the statement in line 23 assigns `'Tails'` to `self.sideup`.

The `get_sideup` method appears in lines 28 through 29:

```
def get_sideup(self):
    return self.sideup
```

Once again, the method has the required `self` parameter variable. This method simply returns the value of `self.sideup`. We call this method any time we want to know which side of the coin is facing up.

To demonstrate the `Coin` class, we need to write a complete program that uses it to create an object. Program 9-2 shows an example. The `Coin` class definition appears in lines 6 through 29. The program has a `main` function, which appears in lines 32 through 44.

Program 9-2 (coin_demo1.py)

```
 1  import random
 2
 3  # The Coin class simulates a coin that can
 4  # be flipped.
 5
 6  class Coin:
 7
 8      # The __init__ method initializes the
 9      # sideup data attribute with 'Heads'.
10
11      def __init__(self):
12          self.sideup = 'Heads'
13
14      # The toss method generates a random number
15      # in the range of 0 through 1. If the number
16      # is 0, then sideup is set to 'Heads'.
17      # Otherwise, sideup is set to 'Tails'.
18
19      def toss(self):
20          if random.randint(0, 1) == 0:
```

(program continues)

Program 9-2 *(continued)*

```
21                 self.sideup = 'Heads'
22             else:
23                 self.sideup = 'Tails'
24 
25         # The get-sideup method returns the value
26         # referenced by sideup.
27 
28         def get_sideup(self):
29             return self.sideup
30 
31 # The main function.
32 def main():
33     # Create an object from the Coin class.
34     my-coin = Coin()
35 
36     # Display the side of the coin that is facing up.
37     print 'This side is up:', my_coin.get_sideup()
38 
39     # Toss the coin.
40     print 'I am tossing the coin...'
41     my_coin.toss()
42 
43     # Display the side of the coin that is facing up.
44     print 'This side is up:', my_coin.get_sideup()
45 
46 # Call the main function.
47 main()
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Tails
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Tails
```

Talte a closer look at the statement in line 34:

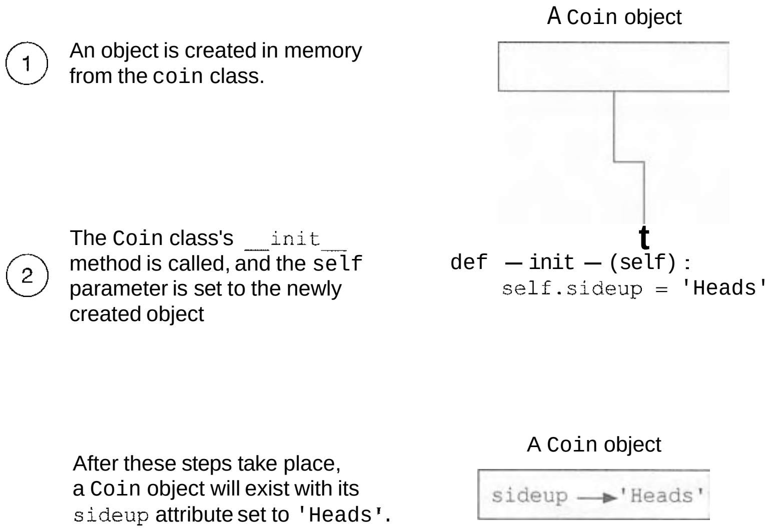
```
my-coin = Coin()
```

The expression `Coin()` that appears on the right side of the = operator causes two things to happen:

1. An object is created in memory from the `Coin` class.
2. The `Coin` class's `__init__` method is executed, and the `self` parameter is automatically set to the object that was just created. As a result, that object's `sideup` attribute is assigned the string `'Heads'`.

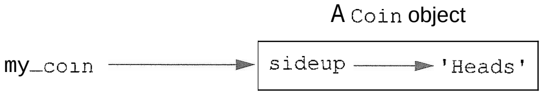
Figure 9-6 illustrates these steps.

Figure 9-6 Actions caused by the `Coin()` expression


A Coin object
①
An object is created in memory from the coin class.
②
The Coin class's __init__ method is called, and the self parameter is set to the newly created object
def __init__(self):
    self.sideup = 'Heads'
After these steps take place, a Coin object will exist with its sideup attribute set to 'Heads'.
A Coin object
sideup
'Heads'

After this, the = operator assigns the `Coin` object that was just created to the `my_coin` variable. Figure 9-7 shows that after the statement in line 12 executes, the `my_coin` variable will reference a `Coin` object, and that object's `sideup` attribute will be assigned the string `'Heads'`.

Figure 9-7 The `my_coin` variable references a `Coin` object


A Coin object
my_coin
sideup
'Heads'

The next statement to execute is line 37:

```
print 'This side is up:', my_coin.get_sideup()
```

This statement prints a message indicating the side of the coin that is facing up. Notice that the following expression appears in the statement:

```
my_coin.get_sideup()
```

This expression uses the object referenced by `my_coin` to call the `get_sideup` method. When the method executes, the `self` parameter will reference the `my_coin` object. As a result, the method returns the string `'Heads'`.

Notice that we did not have to pass an argument to the `sideup` method, despite the fact that it has the `self` parameter variable. When a method is called, Python automatically passes a reference to the calling object into the method's first parameter. As a result, the `self` parameter will automatically reference the object that the method is to operate on.

Lines 40 and 41 are the next statements to execute:

```
print 'I am tossing the coin..,'
my_coin.toss()
```

The statement in line 41 uses the object referenced by `my_coin` to call the `toss` method. When the method executes, the `self` parameter will reference the `my_coin` object. The method will randomly generate a number and use that number to change the value of the object's `sideup` attribute.

Line 44 executes next. This is another `print` statement that calls `my_coin.get_sideup()` to display the side of the coin that is facing up.

Hiding Attributes

Earlier in this chapter we mentioned that an object's data attributes should be private, so that only the object's methods can directly access them. This protects the object's data attributes from accidental corruption. However, in the `Coin` class that was shown in the previous example, the `sideup` attribute is not private. It can be directly accessed by statements that are not in a `Coin` class method. Program 9-3 shows an example. Note that lines 1 through 30 are not shown to conserve space. Those lines contain the `Coin` class, and they are the same as lines 1 through 30 in Program 9-2.

Program ***9-3*** (coin_demo2.py)

Lines 1 through 30 are omitted. These lines are the same as lines 1 through 30 in Program 9-2.

```
31  # The main function.
32  def main():
33      # Create an object from the Coin class.
34      my-coin = Coin()
35
36      # Display the side of the coin that is facing up.
37      print 'This side is up:', my_coin.get_sideup()
38
39      # Toss the coin.
40      print 'I am tossing the coin...'
41      my_coin.toss()
42
43      # But now I'm going to cheat! I'm going to
44      # directly change the value of the object's
45      # sideup attribute to 'Heads'.
46      my_coin.sideup = 'Heads'
47
48      # Display the side of the coin that is facing up.
```

```
49      print 'This side is up:', my_coin.get_sideup()
50
51  # Call the main function.
52  main()
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Program Output

```
This side is up: Heads
I am tossing the coin...
This side is up: Heads
```

Line 34 creates a `Coin` object in memory and assigns it to the `my_coin` variable. The `print` statement in line 37 displays the side of the coin that is facing up, and then line 41 calls the object's `toss` method. Then the statement in line 46 directly assigns the string `'Heads'` to the object's `sideup` attribute:

```
my_coin.sideup = 'Heads'
```

Regardless of the outcome of the `toss` method, this statement will change the `my_coin` object's `sideup` attribute to `'Heads'`. As you can see from the three sample runs of the program, the coin always lands heads up!

If we truly want to simulate a coin that is being tossed, then we don't want code outside the class to be able to change the result of the `toss` method. To prevent this from happening, we need to make the `sideup` attribute private. In Python you can hide an attribute by starting its name with two underscore characters. If we change the name of the `sideup` attribute to `__sideup`, then code outside the `Coin` class will not be able to access it. Program 9-4 shows a new version of the `Coin` class, with this change made.

Program 9-4 (coin.. demo3.py)

```
import random

# The Coin class simulates a coin that can
# be flipped.

class Coin:

    # The __init__ method initializes the
```

(program continues)

Program 9-4 *(continued)*

```
    # __sideup data attribute with 'Heads'.

    def __init--(self):
        self.--sideup = 'Heads'

    # The toss method generates a random number
    # in the range of 0 through 1. If the number
    # is 0, then sideup is set to 'Heads'.
    # Otherwise, sideup is set to 'Tails'.

    def toss(self):
        if random.randint(0, 1) == 0:
            self.__sideup = 'Heads'
        else:
            self.__sideup = 'Tails'

    # The get-sideup method returns the value
    # referenced by sideup.

    def get_sideup(self):
        return self.__sideup

# The main function.
def main():
    # Create an object from the Coin class.
    my-coin = Coin()

    # Display the side of the coin that is facing up.
    print 'This side is up:', my_coin.get_sideup()

    # Toss the coin.
    print 'I am going to toss the coin ten times:'
    for count in range(10):
        my_coin.toss()
        print my-coin.get-sideup()

# Call the main function.
main()
```

Program Output

```
This sid[illegible]: Heads
I am goi[illegible]oss the coin ten times:
Tails
Heads
Heads
```

```
Tails
Tails
Tails
Tails
Tails
Heads
Heads
```

Storing Classes in Modules

The programs you have seen so far in this chapter have the `Coin` class definition in the same file as the programming statements that use the `Coin` class. This approach works fine with small programs that use only one or two classes. As programs use more classes, however, the need to organize those classes becomes greater.

Programmers commonly organize their class definitions by storing them in modules. Then the modules can be imported into any programs that need to use the classes they contain. For example, suppose we decide to store the `Coin` class in a module named `simulation`. Program *9-5* shows the contents of the `simulation.py` file. Then, when we need to use the `Coin` class in a program, we can import the `simulation` module. This is demonstrated in Program 9-6.

Program 9-5 (simulation.py)

```
import random

# The Coin class simulates a coin that can
# be flipped.

class Coin:

    # The __init__ method initializes the
    # __sideup data attribute with 'Heads'.

    def __init--(self):
        self.__sideup = 'Heads'

    # The toss method generates a random number
    # in the range of 0 through 1. If the number
    # is 0, then sideup is set to 'Heads'.
    # Otherwise, sideup is set to 'Tails'.

    def toss(self):
        if random.randint(0, 1) == 0:
            self.__sideup = 'Heads'
        else:
            self.__sideup = 'Tails'
```

(program continues)

Program 9-5 *(continued)*

```

    # The get-sideup method returns the value
    # referenced by sideup.

    def get_sideup(self):
        return self.__sideup
```

Program 9-6 (coin_demo4.py)

```
# This program imports the simulation module and
# creates an instance of the Coin class.

import simulation

def main():
    # Create an object from the Coin class.
    my_coin = simulation.Coin()

    # Display the side of the coin that is facing up.
    print 'This side is up:', my_coin.get_sideup()

    # Toss the coin.
    print 'I am going to toss the coin ten times:'
    for count in range(10):
        my_coin.toss()
        print my_coin.get_sideup()

# Call the main function.
main()
```

Program Output

```
This side is up: Heads
I am going to toss the coin ten times:
Tails
Tails
Heads
Tails
Heads
Heads
Tails
Heads
Tails
Tails
```

Line 4 imports the `simulation` module. Notice that in line 8 we had to qualify the name of the `Circle` class by prefixing it with the name of the module, followed by a dot:

```
my-coin = simulation.Coin()
```

The BankAccount Class

Let's look at another example. Program 9-7 shows a `BankAccount` class, stored in a module named `account`. Objects that are created from this class will simulate bank accounts, allowing us to have a starting balance, make deposits, make withdrawals, and get the current balance.

Program 9-7 (account.py)

```
 1  # The BankAccount class simulates a bank account.
 2
 3  class BankAccount:
 4
 5      # The __init__ method accepts an argument for
 6      # the account's balance. It is assigned to
 7      # the __balance attribute.
 8
 9      def __init--(self, bal):
10          self.--balance = bal
11
12      # The deposit method makes a deposit into the
13      # account.
14
15      def deposit(self, amount):
16          self.--balance += amount
17
18      # The withdraw method withdraws an amount
19      # from the account.
20
21      def withdraw(self, amount):
22          if self.--balance >= amount:
23              self.--balance -= amount
24          else:
25              print 'Error: Insufficient funds'
26
27      # The get-balance method returns the
28      # account balance.
29
30      def get_balance(self):
31          return self.__balance
```

Notice that the `__init__` method has two parameter variables: `self` and `bal`. The `bal` parameter will accept the account's starting balance as an argument. In line 10 the `bal` parameter amount is assigned to the object's `__balance` attribute.

The `deposit` method is in lines 15 through 16. This method has two parameter variables: `self` and `amount`. When the method is called, the amount that is to be deposited into the account is passed into the `amount` parameter. The value of the parameter is then added to the `__balance` attribute in line 16.

The `withdraw` method is in lines 21 through 25. This method has two parameter variables: `self` and `amount`. When the method is called, the amount that is to be withdrawn from the account is passed into the `amount` parameter. The `if` statement that begins in line 22 determines whether there is enough in the account balance to make the withdrawal. If so, amount is subtracted from `__balance` in line 23. Otherwise line 25 displays the message `'Error: Insufficient funds'`.

The `get_balance` method is in lines 30 through 31. This method returns the value of the `__balance` attribute.

Program 9-8 demonstrates how to use the class.

Program 9-8 (account_test.py)

```
# This program demonstrates the BankAccount class.

import account

def main():
    # Get the starting balance.
    start-bal = input('Enter your starting balance: ')

    # Create a BankAccount object.
    savings = account.BankAccount(start_bal)

    # Deposit the user's paycheck.
    pay = input('How much were you paid this week? ')
    print 'I will deposit that into your account.'
    savings.deposit(pay)

    # Display the balance.
    print 'Your account balance is $%.2f.' 8 savings.get_balance()

    # Get the amount to withdraw..
    cash = input('How much would you like to withdraw? ')
    print 'I will withdraw that from your account.'
    savings.withdraw(cash)

    # Display the balance.
    print 'Your account balance is $%.2f.' % savings.get_balance()

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00 [Enter]
How much were you paid this week? 500.00 [Enter]
I will deposit that into your account.
Your account balance is $1500.00.
How much would you like to withdraw? 1200.00 [Enter]
I will withdraw that from your account.
Your account balance is $300.00.
```

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00 [Enter]
How much were you paid this week? 500.00 [Enter]
I will deposit that into your account.
Your account balance is $1500.00.
How much would you like to withdraw? 2000.00 [Enter]
I will withdraw that from your account.
Error: Insufficient funds
Your account balance is $1500.00.
```

Line 7 gets the starting account balance from the user and assigns it to the `start_bal` variable. Line 10 creates an instance of the BankAccount class and assigns it to the `savings` variable. Take a closer look at the statement:

```
savings = account.BankAccount(start_bal)
```

Notice that the `start_bal` variable is listed inside the parentheses. This causes the `start_bal` variable to be passed as an argument to the `__init__` method. In the `__init__` method, it will be passed into the `bal` parameter.

Line 13 gets the amount of the user's pay and assigns it to the `pay` variable. In line 15 the `savings.deposit` method is called, passing the `pay` variable as an argument. In the `deposit` method, it will be passed into the `amount` parameter.

Line 18 displays the account balance. The `print` statement displays the value returned from the `savings.get_balance` method.

Line 21 gets the amount that the user wants to withdraw and assigns it to the `cash` variable. In line 23 the `savings.withdraw` method is called, passing the `cash` variable as an argument. In the `withdraw` method, it will be passed into the `amount` parameter. Line 26 displays the ending account balance.

The `__str__` method

Quite often we need to display a message that indicates an object's state. An object's *state* is simply the values of the object's attributes at any given moment. For example, recall that the `BankAccount` class has one data attribute: `__balance`. At any given moment, a `BankAccount` object's `__balance` attribute will reference some value. The value of the `__balance` attribute represents the object's state at that moment. The following might be an example of code that displays a `BankAccount` object's state:

```
account = BankAccount(1500.0)
print 'Your account balance is $%.2f.' % savings.get_balance()
```

The first statement creates a `BankAccount` object, passing the value 1500.0 to the `__init__` method. After this statement executes, the `account` variable will reference the `BankAccount` object. In the second line, the `print` statement displays a string showing the value of the object's `__balance` attribute. The output of this statement will look like this:

```
Your account balance is $1500.00.
```

Displaying an object's state is a common task. It is so common that many programmers equip their classes with a method that returns a string containing the object's state. In Python, you give this method the special name `__str__`. Program 9-9 shows the `BankAccount` class with a `__str__` method added to it. The `__str__` method appears in lines 36 through 39. It creates a string containing the account balance, and returns that string.

Program 9-9 (account2.py)

```
 1  # The BankAccount class simulates a bank account.
 2
 3  class BankAccount:
 4
 5      # The __init__ method accepts an argument for
 6      # the account's balance. It is assigned to
 7      # the --balance attribute.
 8
 9      def __init__(self, bal):
10          self.--balance = bal
11
12      # The deposit method makes a deposit into the
13      # account.
14
15      def deposit(self, amount):
16          self.--balance += amount
17
18      # The withdraw method withdraws an amount
19      # from the account.
20
2I      def withdraw(self, amount):
22          if self.--balance >= amount:
23              self.__balance -= amount
24          else:
25              print 'Error: Insufficient funds'
26
27      # The get-balance method returns the
28      # account balance.
29
30      def get_balance(self):
31          return self.--balance
32
33      # The __str__ method returns a string
34      # indicating the object's state.
```

```

    def __str__(self):
        state-string = 'The account balance is $%.2f.' \
                       % self.--balance
        return state_string
```

You do not directly call the `__str__` method. Instead, it is automatically called when you pass the object's name to the `print` statement. Program 9-10 shows an example.

Program 9-10 (account_test2.py)

```
# This program demonstrates the BankAccount class
# with the __str__ method added to it.

import account2

def main():
    # Get the starting balance.
    start-bal = input('Enter your starting balance: ')

    # Create a BankAccount object.
    savings = account2.BankAccount(start_bal)

    # Deposit the user's paycheck.
    pay = input('How much were you paid this week? ')
    print 'I will deposit that into your account.'
    savings.deposit(pay)

    # Display the balance.
    print savings

    # Get the amount to withdraw.
    cash = input('How much would you like to withdraw? ')
    print 'I will withdraw that from your account.'
    savings.withdraw(cash)

    # Display the balance.
    print savings

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter your starting balance: 1000.00[Enter]
How much were you paid this week? 500.00 [Enter]
I will deposit that into your account.
```

(program output continues)

Program Output *(continued)*

```
The account balance is $1500.00.
How much would you like to withdraw? 1200.00 [Enter]
I will withdraw that from your account.
The account balance is $300.00.
```

The name of the object, `savings`, is passed to the `print` statements in lines 19 and 27. This causes the `BankAccount` class's `__str__` method to be called. The string that is returned from the `__str__` method is then displayed.

The `__str__` method is also called automatically when an object is passed as an argument to the built-in `str` function. Here is an example:

```
account = BankAccount(1500.0)
message = str(account)
print message
```

In the second statement, the `account` object is passed as an argument to the `str` function. This causes the `BankAccount` class's `__str__` method to be called. The string that is returned is assigned to the `message` variable and then displayed by the `print` statement in the third line.

Checkpoint

9.5 You hear someone make the following comment: "A blueprint is a design for a house. A carpenter can use the blueprint to build the house. If the carpenter wishes, he or she can build several identical houses from the same blueprint." Think of this as a metaphor for classes and objects. Does the blueprint represent a class, or does it represent an object?

9.6 In this chapter, we use the metaphor of a cookie cutter and cookies that are made from the cookie cutter to describe classes and objects. In this metaphor, are objects the cookie cutter, or the cookies?

9.7 What is the purpose of the `__init__` method? When does it execute?

9.8 What is the purpose of the `self` parameter in a method?

9.9 In a Python class, how do you hide an attribute from code outside the class?

9.10 What is the purpose of the `__str__` method?

9.11 How do you call the `__str__` method?

9.3 Working with instances

CONCEPT: Each instance of a class has its own set of data attributes.

When a method uses the `self` parameter to create an attribute, the attribute belongs to the specific object that `self` references. We call these attributes *instance attributes*, because they belong to a specific instance of the class.

It is possible to create many instances of the same class in a program. Each instance will then have its own set of attributes. For example, look at Program 9-11. This program creates three instances of the `Coin` class. Each instance has its own `__sideup` attribute.

Program 9-11 (coin_demo5.py)

```
# This program imports the simulation module and
# creates three instances of the Coin class.

import simulation

def main():
    # Create three objects from the Coin class.
    coin1 = simulation.Coin()
    coin2 = simulation.Coin()
    coin3 = simulation.Coin()

    # Display the side of each coin that is facing up.
    print 'I have three coins with these sides up:'
    print coin1.get_sideup()
    print coin2.get_sideup()
    print coin3.get_sideup()
    print

    # Toss the coin.
    print 'I am tossing all three coins...'
    print
    coin1.toss()
    coin2.toss()
    coin3.toss()

    # Display the side of each coin that is facing up.
    print 'Now here are the sides that are up:'
    print coin1.get_sideup()
    print coin2.get_sideup()
    print coin3.get_sideup()
    print

# Call the main function.
main()
```

Program Output

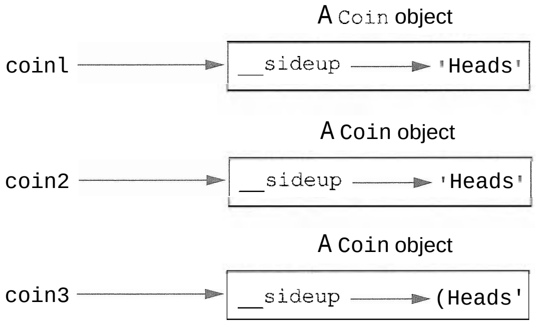
```
I have three coins with these sides up:
Heads
Heads
Heads

I am tossing all three coins...

Now here are the sides that are up:
Tails
Tails
Heads
```

In lines 8 through 10, the following statements create three objects, each an instance of the `Coin` class:

```
coin1 = simulation.Coin()
coin2 = simulation.Coin()
coin3 = simulation.Coin()
```

Figure 9-8 illustrates how the `coin1`, `coin2`, and `coin3` variables reference the three objects after these statements execute. Notice that each object has its own `__sideup` attribute. Lines 14 through 16 display the values returned from each object's `get_sideup` method.

Figure 9-8 The `coin1`, `coin2`, and `coin3` variables reference three `Coin` objects

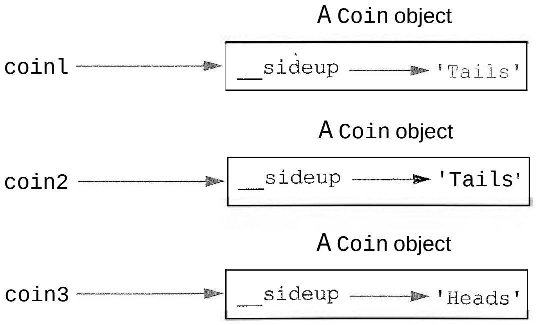

A Coin object
coin1
__sideup
'Heads'
A Coin object
coin2
__sideup
'Heads'
A Coin object
coin3
__sideup
'Heads'

Then, the statements in lines 22 through 24 call each object's `toss` method:

```
coin1.toss()
coin2.toss()
coin3.toss()
```

Figure 9-9 shows how these statements changed each object's `__sideup` attribute in the program's sample run.

Figure 9-9 The objects after the `toss` method


A Coin object
coin1
__sideup
'Tails'
A Coin object
coin2
__sideup
'Tails'
A Coin object
coin3
__sideup
'Heads'

In the Spotlight:

Creating the `CellPhone` Class

Wireless Solutions, Inc. is a business that sells cell phones and wireless service. You are a programmer in the company's IT department, and your team is designing a program to manage all of the cell phones that are in inventory. You have been asked to design a class that represents a cell phone. The data that should be kept as attributes in the class are as follows:

- The name of the phone's manufacturer will be assigned to the `__manufact` attribute.
- The phone's model number will be assigned to the `__model` attribute.
- The phone's retail price will be assigned to the `__retail_price` attribute.

The class will also have the following methods:

- An `__init__` method that accepts arguments for the manufacturer, model number, and retail price.
- A `set_manufact` method that accepts an argument for the manufacturer. This method will allow us to change the value of the `__manufact` attribute after the object has been created, if necessary.
- A `set_model` method that accepts an argument for the model. This method will allow us to change the value of the `__model` attribute after the object has been created, if necessary.
- A `set_retail_price` method that accepts an argument for the retail price. This method will allow us to change the value of the `__retail_price` attribute after the object has been created, if necessary.
- A `get_manufact` method that returns the phone's manufacturer.
- A `get_model` method that returns the phone's model number.
- A `get_retail_price` method that returns the phone's retail price.

Program 9-12 shows the class definition. The class is stored in a module named `inventory`.

Program 9-12 (inventory.py)

```python
# The CellPhone class holds data about a cell phone.

class CellPhone:

    # The __init__ method initializes the attributes.

    def __init__(self, manufact, model, price):
        self.__manufact = manufact
        self.__model = model
        self.__retail_price = price

    # The set_manufact method accepts an argument for
    # the phone's manufacturer.

    def set_manufact(self, manufact):
        self.__manufact = manufact
```

(program continues)

Program 9-12 *(continued)*

```

    # The set_model method accepts an argument for
    # the phone's model number.

    def set_model(self, model):
        self.__model = model

    # The set_retail_price method accepts an argument
    # for the phone's retail price.

    def set_retail_price(self, price):
        self.__retail_price = price

    # The get_manufact method returns the
    # phone's manufacturer.

    def get_manufact(self):
        return self.__manufact

    # The get_model method returns the
    # phone's model number.

    def get_model(self):
        return self.__model

    # The get_retail_price method returns the
    # phone's retail price.

    def get_retail_price(self):
        return self.__retail_price
```

The `CellPhone` class will be imported into several programs that your team is developing. To test the class, you write the code in Program 9-13. This is a simple program that prompts the user for the phone's manufacturer, model number, and retail price. An instance of the `CellPhone` class is created and the data is assigned to its attributes.

Program 9-13 (cell_phone_test.py)

```
# This program tests the CellPhone class.

import inventory

def main():
    # Get the phone data.
    man = raw_input('Enter the manufacturer: ')
    mod = raw_input('Enter the model number: ')
    retail = input('Enter the retail price: ')

    # Create an instance of the CellPhone class.
    phone = inventory.CellPhone(man, mod, retail)

    # Display the data that was entered.
    print 'Here is the data that you entered:'
    print 'Manufacturer:', phone.get_manufact()
    print 'Model Number:', phone.get_model()
    print 'Retail Price: $%.2f' % phone.get_retail_price()

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]
Here is the data that you entered:
Manufacturer: Acme Electronics
Model Number: M1000
Retail Price: $199.99
```

Assessor and Mutator Methods

As mentioned earlier, it is a common practice to make all of a class's data attributes private and to provide public methods for accessing and changing those attributes. This ensures that the object owning those attributes is in control of all the changes being made to them.

A method that returns a value from a class's attribute but does not change it is known as an *accessor method*. Accessor methods provide a safe way for code outside the class to retrieve the values of attributes, without exposing the attributes in a way that they could be changed by the code outside the method. In the `CellPhone` class that you saw in Program 9-12 (in the previous In *the Spotlight* section), the `get_manufact`, `get_model`, and `get_retail_price` methods are accessor methods.

A method that stores a value in a data attribute or changes the value of a data attribute in some other way is known as a *mutator method*. Mutator methods can control the way that a class's data attributes are modified. When code outside the class needs to change the value of an object's data attribute, it typically calls a mutator and passes the new value as an argument. If necessary, the mutator can validate the value before it assigns it to the data attribute. In Program 9-12, the `set_manufact`, `set_model`, and `set_retail_price` methods are mutator methods.

NOTE: Mutator methods are sometimes called "setters" and accessor methods are sometimes called "getters."

In the Spotlight:

Storing Objects in a List

The `CellPhone` class that you created in the previous *In the Spotlight* section will be used in a variety of programs. Many of these programs will store `CellPhone` objects in lists. To test the ability to store `CellPhone` objects in a list, you write the code in Program 9-14. This program gets the data for five phones from the user, creates five `CellPhone` objects holding that data, and stores those objects in a list. It then iterates over the list displaying the attributes of each object.

Program 9-14 (cell_phone_list.py)

```
# This program creates five CellPhone objects and
# stores them in a list.

import inventory

def main():
    # Get a list of CellPhone objects.
    phones = make_list()

    # Display the data in the list.
    print 'Here is the data you entered:'
    display_list(phones)

# The make_list function gets data from the user
# for five phones. The function returns a list
# of CellPhone objects containing the data.

def make_list():
    # Create an empty list.
    phone_list = []

    # Add five CellPhone objects to the list.
    print 'Enter data for five phones.'
    for count in range(1, 6):
        # Get the phone data.
        print 'Phone number ' + str(count) + ':'
        man = raw_input('Enter the manufacturer: ')
        mod = raw_input('Enter the model number: ')
        retail = input('Enter the retail price: ')
        print

        # Create a new CellPhone object in memory and
        # assign it to the phone variable.
        phone = inventory.CellPhone(man, mod, retail)

        # Add the object to the list.
        phone_list.append(phone)

    # Return the list.
    return phone_list

# The display_list function accepts a list containing
# CellPhone objects as an argument and displays the
# data stored in each object.

def display_list(phone_list):
    for item in phone_list:
        print item.get_manufact()
        print item.get_model()
        print item.get_retail_price()
        print

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter data for five phones.

Phone number 1:
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]

Phone number 2:
Enter the manufacturer: Atlantic Communications [Enter]
Enter the model number: S2 [Enter]
Enter the retail price: 149.99 [Enter]

Phone number 3:
Enter the manufacturer: Wavelength Electronics [Enter]
Enter the model number: N477 [Enter]
Enter the retail price: 249.99 [Enter]

Phone number 4:
Enter the manufacturer: Edison Wireless [Enter]
Enter the model number: SLX88 [Enter]
Enter the retail price: 169.99 [Enter]

Phone number 5:
Enter the manufacturer: Sonic Systems [Enter]
Enter the model number: X99 [Enter]
Enter the retail price: 299.99 [Enter]

Here is the data you entered:
Acme Electronics
M1000
199.99
```

(program output continues)

Program Output *(continued)*

```
Atlantic Communications
S2
149.99

Wavelength Electronics
N477
249.99

Edison Wireless
SLX88
169.99

Sonic Systems
X99
299.99
```

The `make_list` function appears in lines 18 through 40. In line 20 an empty list named `phone_list` is created. The `for` loop, which begins in line 24, iterates five times. Each time the loop iterates, it gets the data for a cell phone from the user (lines 27 through 29), it creates an instance of the `CellPhone` class that is initialized with the data (line 34), and it appends the object to the `phone_list` list (line 37). Line 40 returns the list.

The `display_list` function in lines 46 through 51 accepts a list of `CellPhone` objects as an argument. The `for` loop that begins in line 47 iterates over the objects in the list and displays the values of each object's attributes.

Checkpoint

9.12 What is an instance attribute?

9.13 A program creates 10 instances of the `Coin` class. How many `__sideup` attributes exist in memory?

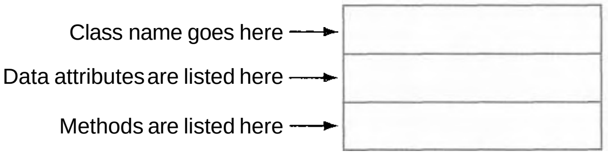
9.14 What is an accessor method? What is a mutator method?

9.4 Techniques for Designing Classes

The Unified Modeling Language

When designing a class, it is often helpful to draw a UML diagram. UML stands for Unified Modeling Language. It provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 9-10 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's data attributes. The bottom section holds a list of the class's methods.

Figure 9-10 General layout of a UML diagram for a class


Class name goes here
Data attributes are listed here
Methods are listed here

Following this layout, Figure 9-11 and 9-12 show UML diagrams for the `Coin` class and the `CellPhone` class that you saw previously in this chapter. Notice that we did not show the `self` parameter in any of the methods, since it is understood that the `self` parameter is required.

Figure 9-11 UML diagram for the `Coin` class

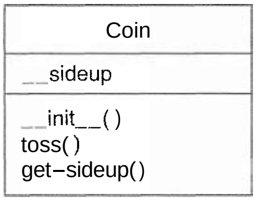
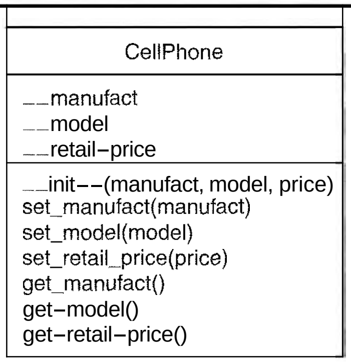

Coin
__sideup
__init__()
toss()
get-sideup()

Figure 9-12 UML diagram for the `CellPhone` class


CellPhone
__manufact
__model
__retail-price
__init--(manufact, model, price)
set_manufact(manufact)
set_model(model)
set_retail_price(price)
get_manufact()
get-model()
get-retail-price()

Finding the Classes in a Problem

When developing an object-oriented program, one of your first tasks is to identify the classes that you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem, and then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps.

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

Writing a Description of the Problem Domain

The problem domain is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are writing a program that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

> Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher, student, etc.
- The results of a business event, such as a customer order, or in this case a service quote
- Recordkeeping items, such as customer histories and payroll records

identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

> Joe's Automotive Shop services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

Notice that some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them.

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
foreign cars
Joe's Automotive Shop
make
manager
Mercedes
model
name

Porsche
sales tax,
service quote
shop
telephone number
total estimated charges
year

Refining the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

1. Some of the nouns really mean the same thing.

In this example, the following sets of nouns refer to the same thing:

- cars and foreign cars
  These all refer to the general concept of a car.
- Joe's Automotive Shop and shop
  Both of these refer to the company "Joe's Automotive Shop."

We can settle on a single class for each of these. In this example we will arbitrarily eliminate foreign cars from the list, and use the word cars. Likewise we will eliminate Joe's Automotive Shop from the list and use the word shop. The updated list of potential classes is:

---

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
manager
Mercedes
model
name
Porsche
sales tax
service quote

Because cars and foreign cars mean the same thing in this problem, we have eliminated foreign cars. Also, because Joe's Automotive Shop and shop mean the same thing, we have eliminated Joe's Automotive Shop.

(continued)

shop
telephone number
total estimated charges
year

2. Some nouns might represent items that we do not need to be concerned with in order to solve the problem.

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example we can eliminate two unnecessary classes from the list:

- We can cross shop off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the manager because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
~~manager~~
Mercedes
model
name
Porsche
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

Our problem description does not direct us to process any information about the shop, or any information about the manager, so we have eliminated those from the list.

3. Some of the nouns might represent objects, not classes.

We can eliminate Mercedes, Porsche, and BMW as classes because, in this example, they all represent specific cars, and can be considered instances of a cars class. Also, we can eliminate the word car from the list. In the description it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a cars class. At this point the updated list of potential classes is:

---

address
~~BMW~~
~~car~~
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's~~
~~manager~~
make
~~Mercedes~~
model
name
~~Porsche~~
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

We have eliminated Mercedes, Porsche, BMW, and car because they are all instances of a cars class. That means that these nouns identify objects, not classes.

NOTE: Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class and a singular noun will indicate an object.

4. Some of the nouns might represent simple values that can be assigned to a variable and do not require a class.

Remember, a class contains data attributes and methods. Data attributes are related items that are stored in an object of the class, and define the object's state. Methods are actions or behaviors that can be performed by an object of the class. If a noun represents a type of item that would not have any identifiable data attributes or methods, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have data attributes and methods, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges and year. These are all simple string or numeric values that can be stored in variables. Here is the updated list of potential classes:

~~Address~~
~~BMW~~
~~car~~
cars
customer
~~estimated labor charges~~
~~estimated parts charges~~
~~foreign cars~~
~~Joe's Automotive Shop~~
~~make~~
~~manager~~
~~Mercedes~~
~~model~~
~~name~~
~~Porsche~~
~~sales tax~~
service quote
~~shop~~
~~telephone number~~
~~total estimated charges~~
~~year~~

We have eliminated address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges, and year as classes because they represent simple values that can be stored in variables.

As you can see from the list, we have eliminated everything except cars, customer, and service quote. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

When you have identified the things that a class is responsible for knowing, then you have identified the class's data attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its methods.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so further consideration is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

The `Customer` Class

In the context of our problem domain, what must the `customer` class know? The description directly mentions the following items, which are all data attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number

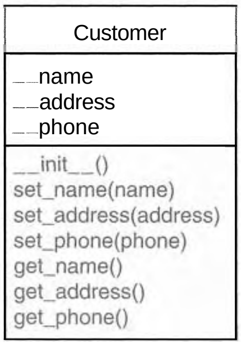
These are all values that can be represented as strings and stored as data attributes. The `customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's methods. In the context of our problem domain, what must the `customer` class do? The only obvious actions are:

- initialize an object of the `customer` class
- set and return the customer's name
- set and return the customer's address
- set and return the customer's telephone number

From this list we can see that the `customer` class will have an `__init__` method, as well as accessors and mutators for the data attributes. Figure 9-13 shows a UML diagram for the `customer` class.

Figure 9-13 UML diagram for the `Customer` class



The `Car` Class

In the context of our problem domain, what must an object of the `Car` class know? The following items are all data attributes of a car, and are mentioned in the problem domain:

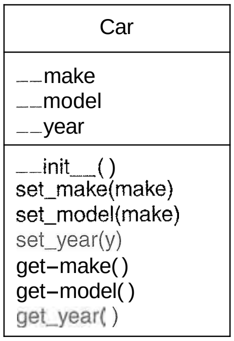
- the car's make
- the car's model
- the car's year

Now let's identify the class's methods. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of methods that we will find in most classes (an `__init__` method, accessors, and mutators). Specifically, the actions are:

- initialize an object of the **`Car`** class
- set and get the car's make
- set and get the car's model
- set and get the car's year

Figure 9-14 shows a UML diagram for the **`Car`** class at this point.

Figure 9-14 UML diagram for the `Car` class



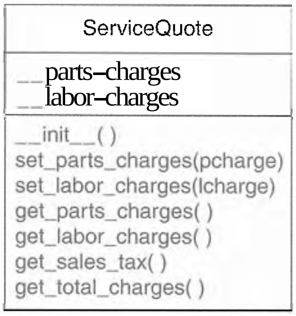
The `ServiceQuote` Class

In the context of our problem domain, what must an object of the **`ServiceQuote`** class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges
- the sales tax
- the total estimated charges

The methods that we will need for this class are an `__init__` method and the accessors and mutators for the estimated parts charges and estimated labor charges attributes. In addition, the class will need methods that calculate and return the sales tax and the total estimated charges. Figure 9-15 shows a UML diagram for the **`ServiceQuote`** class.

Figure 8-15 UML diagram for the ServiceQuote class


ServiceQuote
__parts–charges
__labor–charges
__init__( )
set_parts_charges(pcharge)
set_labor_charges(lcharge)
get_parts_charges( )
get_labor_charges( )
get_sales_tax( )
get_total_charges( )

This is Only the Beginning

You should look at the process that we have discussed in this section merely as a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.

Checkpoint

9.15 The typical UML diagram for a class has three sections. What appears in these three sections?

9.16 What is a problem domain?

9.17 When designing an object-oriented application, who should write a description of the problem domain?

9.18 How do you identify the potential classes in a problem domain description?

9.19 What are a class's responsibilities?

9.20 What two questions should you ask to determine a class's responsibilities?

9.21. Will all of a classes actions always be directly mentioned in the problem domain description?

Review Questions

Multiple Choice

1. The ______________ programming practice is centered on creating functions that are separate from the data that they work on.
   a. modular
   b. procedural
   c. functional
   d. object-oriented

2. The ____________ programming practice is centered on creating objects.
   a. object-centric
   b. objective
   c. procedural
   d. object-oriented

3. A(n) ____________ is a component of a class that references data.
   a. method
   b. instance
   c. data attribute
   d. module

4. An object is a(n) ____________.
   a. blueprint
   b. cookie cutter
   c. variable
   d. instance

5. By doing this you can hide a class's attribute from code outside the class.
   a. avoid using the `self` parameter to create the attribute
   b. begin the attribute's name with two underscores
   c. begin the name of the attribute with `private__`
   d. begin the name of the attribute with the `@` symbol

6. A(n) ____________ method gets the value of a data attribute but does not change it.
   a. retriever
   b. constructor
   c. mutator
   d. accessor

7. A(n) ____________ method stores a value in a data attribute or changes its value in some other way.
   a. modifier
   b. constructor
   c. mutator
   d. accessor

8. The ____________ method is automatically called when an object is created.
   a. `__init__`
   b. `init`
   c. `__str__`
   d. `__object__`

9. If a class has a method named `__str__`, which of these is a way to call the method?
   a. you call it like any other method: `object.__str__()`
   b. by passing an instance of the class to the built in `str` function
   c. the method is automatically called when the object is created
   d. by passing an instance of the class to the built-in `state` function

10. A set of standard diagrams for graphically depicting object-oriented systems is provided by ______________.
    a. the Unified Modeling Language
    b. flowcharts
    c. pseudocode
    d. the Object Hierarchy System
11. In one approach to identifying the classes in a problem, the programmer identifies the ______________ in a description of the problem domain.
    a. verbs
    b. adjectives
    c. adverbs
    d. nouns
12. In one approach to identifying a class's data attributes and methods, the programmer identifies the class's ______________.
    a. responsibilities
    b. name
    c. synonyms
    d. nouns

True or False

1. The practice of procedural programming is centered on the creation of objects.
2. Object reusability has been a factor in the increased use of object-oriented programming.
3. It is a common practice in object-oriented programming to make all of a class's data attributes accessible to statements outside the class.
4. A class method does not have to have a `self` parameter.
5. Starting an attribute name with two underscores will hide the attribute from code outside the class.
6. You cannot directly call the `__str__` method.
7. One way to find the classes needed for an object-oriented program is to identify all of the verbs in a description of the problem domain.

Short Answer

1. What is encapsulation?
2. Why should an object's data attributes be hidden from code outside the class?
3. What is the difference between a class and an instance of a class?
4. The following statement calls an object's method. What is the name of the method? What is the name of the variable that references the object?

```
wallet.get_dollar()
```

5. When the `__init__` method executes, what does the `self` parameter reference?
6. In a Python class, how do you hide an attribute from code outside the class?
7. How do you call the `__str__` method?

Algorithm Workbench

1. Suppose `my_car` is the name of a variable that references an object, and `go` is the name of a method. Write a statement that uses the `my_car` variable to call the `go` method. (You do not have to pass any arguments to the `go` method.)
2. Write a class definition named `Book`. The `Book` class should have data attributes for a book's title, the author's name, and the publisher's name. The class should also have the following:
   a. An `__init__` method for the class. The method should accept an argument for each of the data attributes.
   b. Accessor and mutator methods for each data attribute.
   c. An `__str__` method that returns a string indicating the state of the object.
3. Look at the following description of a problem domain:

   > The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

   Assume that you are writing a program that will calculate the amount of interest earned for a bank account.

   a. Identify the potential classes in this problem domain.
   b. Refine the list to include only the necessary class or classes for this problem.
   c. Identify the responsibilities of the class or classes.

Programming Exercises

1. `Pet` Class

Write a class named `Pet`, which should have the following data attributes:

- `__name` (for the name of a pet)
- `__animal_type` (for the type of animal that a pet is. Example values are 'Dog', 'Cat', and 'Bird')
- `__age` (for the pet's age)

The `Pet` class should have an `__init__` method that creates these attributes. It should also have the following methods:

- `set_name`
  This method assigns a value to the `__name` field.
- `set_animal_type`
  This method assigns a value to the `__animal_type` field.
- `set_age`
  This method assigns a value to the `__age` field.
- `get_name`
  This method returns the value of the `name` field.
- `get_type`
  This method returns the value of the `type` field.

- `get_age`
  This method returns the value of the `age` field.

Once you have written the class, write a program that creates an object of the class and prompts the user to enter the name, type, and age of his or her pet. This data should be stored as the object's attributes. Use the object's accessor methods to retrieve the pet's name, type, and age and display this data on the screen.

2. `Car` Class

Write a class named `Car` that has the following data attributes:

- `__year_model` (for the car's year model)
- `__make` (for the make of the car)
- `__speed` (for the car's current speed)

The `Car` class should have an `__init__` method that accept the car's year model and make as arguments. These values should be assigned to the object's `__year_model` and `__make` data attributes. It should also assign 0 to the `__speed` data attribute.

The class should also have the following methods:

- `accelerate`
  The `accelerate` method should add 5 to the speed data attribute each time it is called.
- `brake`
  The `brake` method should subtract 5 from the speed data attribute each time it is called.
- `get_speed`
  The `get_speed` method should return the current speed.

Next, design a program that creates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

3. Personal `Information` Class

Design a class that holds the following personal data: name, address, age, and phone number. Write appropriate accessor and mutator methods. Also, write a program that creates three instances of the class. One instance should hold your information, and the other two should hold your friends' or family members' information.

4. Employee Class

Write a class named `Employee` that holds the following data about an employee in attributes: name, ID number, department, and job title.

Once you have written the class, write a program that creates three `Employee` objects to hold the following data:

Name ID	Number	Department	Job Title
Susan Meyers	47899	Accounting	Vice President
Mark Jones	39119	IT	Programmer
Joy Rogers	81774	Manufacturing	Engineer

The program should store this data in the three objects and then display the data for each employee on the screen.

5. `RetailItem` Class

Write a class named `RetailItem` that holds data about an item in a retail store. The class should store the following data in attributes: item description, units in inventory, and price.

Once you have written the class, write a program that creates three `RetailItem` objects and stores the following data in them:

	Description	Units in Inventory	Price
Item #1	Jacket	12	59.95
Item #2	Designer Jeans	40	34.95
Item #3	Shirt	20	24.95

CHAPTER 10 Inheritance

TOPICS

10.1 Introduction to Inheritance
10.2 Polymorphism

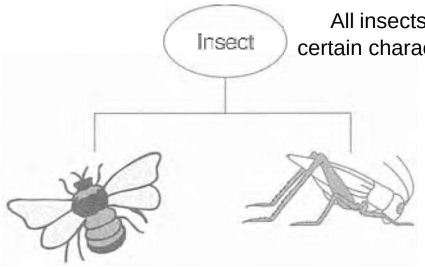
10.1 Introduction to Inheritance

CONCEPT: Inheritance allows a new class to extend an existing class. The new class inherits the members of the class it extends.

Generalization and Specialization

In the real world, you can find many objects that are specialized versions of other more general objects. For example, the term "insect" describes a general type of creature with various characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 10-1.

Figure 10-1 Bumblebees and grasshoppers are specialized versions of an insect



All insects have certain characteristics.

In addition to the common insect characteristics, the bumblebee has its own unique characteristics such as the ability to sting.

In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an "is a" relationship between them. For example, a grasshopper is an insect. Here are a few other examples of the "is a" relationship:

- A poodle is a dog.
- A car is a vehicle.
- A flower is a plant.
- A rectangle is a shape.
- A football player is an athlete.

When an "is a" relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class and the *subclass* is the specialized class. You can think of the subclass as an extended version of the superclass. The subclass inherits attributes and methods from the superclass without any of them having to be rewritten. Furthermore, new attributes and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

NOTE: Superclasses are also called *base classes*, and subclasses are also called *derived classes*. Either set of terms is correct. For consistency, this text will use the terms superclass and subclass.

Let's look at an example of how inheritance can be used. Suppose we are developing a program that a car dealership can use to manage its inventory of used cars. The dealership's inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A `Car` class with data attributes for the make, year model, mileage, price, and the number of doors.
- A **`Truck`** class with data attributes for the make, year model, mileage, price, and the drive type.
- An `SUV` class with data attributes for the make, year model, mileage, price, and the passenger capacity.

This would be an inefficient approach, however, because all three of the classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an **`Automobile`** superclass to hold all the general data about an automobile and then write subclasses for each specific type of automobile. Program 10-1 shows the **`Automobile`** class's code, which appears in a module named **`vehicles`**.

Program 10-1 (Lines 1 through 44 of vehicles.py)

```
# The Automobile class holds general data
# about an automobile in inventory.

class Automobile:
    # The __init__method accepts arguments for the
    # make, model, mileage, and price. It initializes
    # the data attributes with these values.

    def __init__(self, make, model, mileage, price):
        self.__make = make
```

(program continues)

Program 10-1 *(continued)*

```
11          self.__model = model
12          self.--mileage = mileage
13          self.__price = price
14  
15      # The following methods are mutators for the
16      # class's data attributes.
17  
18      def set_make(self, make):
19          self.__make = make
20  
21      def set_model(self, model):
22          self.__model = model
23  
24      def set_mileage(self, mileage):
25          self.--mileage = mileage
26  
27      def set_price(self, price):
28          self.__price = price
29  
30      # The following methods are the accessors
31      # for the class's data attributes.
32  
33      def get_make(self):
34          return self.__make
35  
36      def get_model(self):
37          return self.--model
38  
39      def get_mileage(self):
40          return self.--mileage
41  
42      def get_price(self):
43          return self.__price
44  
```

The Automobile class's `__init__` method accepts arguments for the vehicle's make, model, mileage, and price. It uses those values to initialize the following data attributes:

- `__make`
- `__model`
- `__mileage`
- `__price`

(Recall from Chapter 9 that a data attribute becomes hidden when its name begins with two underscores.) The methods that appear in lines 18 through 28 are mutators for each of the data attributes, and the methods in lines 33 through 43 are the accessors.

The `Automobile` class is a complete class that we can create objects from. If we wish, we can write a program that imports the `vehicle` module and creates instances of the `Automobile` class. However, the `Automobile` class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles we will write subclasses that inherit from the `Automobile` class. Program 10-2 shows the code for the `Car` class, which is also in the `vehicles` module.

Program 10-2 (Lines 45 through 72 of vehicles.py)

```
45  # The Car class represents a car. It is a subclass
46  # of the Automobile class.
47
48  class Car(Automobile):
49      # The __init__ method accepts arguments for the
50      # car's make, model, mileage, price, and doors.
51
52      def __init--(self, make, model, mileage, price, doors):
53          # Call the superclass's __init__ method and pass
54          # the required arguments. Note that we also have
55          # to pass self as an argument.
56          Automobile.--init--(self, make, model, mileage, price)
57
58          # Initialize the __doors attribute.
59          self.__doors = doors
60
63      # The set-doors method is the mutator for the
62      # __doors attribute.
63
64      def set_doors(self, doors):
65          self.__doors = doors
66
67      # The get-doors method is the accessor for the
68      # __doors attribute.
69
70      def get_doors(self):
71          return self.__doors
72
```

Take a closer look at the first line of the class declaration, in line 48:

```
class Car(Automobile):
```

This line indicates that we are defining a class named `Car`, and it inherits from the `Automobile` class. The `Car` class is the subclass and the `Automobile` class is the superclass. If we want to express the relationship between the `Car` class and the `Automobile` class, we can say that a `Car` is an `Automobile`. Because the `Car` class extends the `Automobile` class, it inherits all of the methods and data attributes of the `Automobile` class.

Look at the header for the `__init__` method in line 52:

```
def __init--(self, make, model, mileage, price, doors):
```

Notice that in addition to the required `self` parameter, the method has parameters named make, `model`, `mileage`, `price`, and `doors`. This makes sense because a `Car` object will have data attributes for the car's make, model, mileage, price, and number of doors. Some of these attributes are created by the `Automobile` class, however, so we need to call the `Automobile` class's `__init__` method and pass those values to it. That happens in line 56:

```
Automobile.--init--(self, make, model, mileage, price)
```

This statement calls the `Automobile` class's `__init__` method. Notice that the statement passes the `self` variable, as well as the make, `model`, `mileage`, and `price` variables as arguments. When that method executes, it initializes the `__make`, `__model`, `__mileage`, and `__price` data attributes. Then, in line 59, the `__doors` attribute is initialized with the value passed into the `doors` parameter:

```
self.__doors = doors
```

The `set_doors` method, in lines 64 through 65, is the mutator for the `doors` attribute, and the `get_doors` method, in lines 70 through 71 is the accessor for the'`__doors` attribute. Before going any further, let's demonstrate the `Car` class, as shown in Program 10-3.

Program 10-3 (car_demo.py)

```
# This program demonstrates the Car class.

import vehicles

def main():
    # Create an object from the Car class. ,
    # The car is a 2007 Audi with 12,500 miles, priced
    # at $21,500.00, and has 4 doors.
    used-car = vehicles.Car('Audi', 2007, 12500, 21500.00, 4)

    # Display the car's data.
    print 'Make:', used_car.get_make()
    print 'Model:', used_car.get_model()
    print 'Mileage:', used_car.get_mileage()
    print 'Price:', used_car.get_price()
    print 'Number of doors:', used_car.get_doors()

# Call the main function.
main()
```

Program Output

```
lYake: Audi
lYodel: 2007
```

```
Mileage: 12500
price: 21500.0
Number of doors: 4
```

Line **3** imports the vehicles module, which contains the class definitions for the Automobile and Car classes. Line 9 creates an instance of the Car class, passing 'Audi' as the car's make, 2007 as the car's model, 12500 as the mileage, 21500.00 as the car's price, and 4 as the number of doors. The resulting object is assigned to the used_car variable.

The print statement in lines 12 through 15 calls the object's get_make, get_model, get_mileage, and get_price methods. Even though the Car class does not have any of these methods, it inherits them from the Automobile class. Line 16 calls the get_doors method, which is defined in the Car class.

Now let's look at the Truck class, which also inherits from the Automobile class. The code for the Truck class, which is also in the vehicles module, is shown in Program 10-4.

Program 10-4 (Lines 73 through 100 of vehicles.py)

```
 73 # The Truck class represents a pickup truck. It is a
 74 # subclass of the Automobile class.
 75
 76 class Truck(Automobile):
 77     # The __init__ method accepts arguments for the
 78     # Truck's make, model, mileage, price, and drive type.
 79
 80     def __init--(self, make, model, mileage, price, drive-type):
 81         # Call the superclass's __init__ method and pass
 82         # the required arguments. Note that we also have
 83         # to pass self as an argument.
 84         Automobile.--init--(self, make, model, mileage, price)
 85
 86         # Initialize the __drive_type attribute.
 87         self.--drive-type = drive-type
 88
 89     # The set-drive-type method is the mutator for the
 90     # __drive_type attribute.
 91
 92     def set_drive_type(self, drive-type):
 93         self.__drive = drive-type
 94
 95     # The get-drive-type method is the accessor for the
 96     # __drive_type attribute.
 97
 98     def get_drive_type(self):
 99         return self.--drive-type
100
```

The `Truck` class's `__init__` method begins in line 80. Notice that it takes arguments for the truck's make, model, mileage, price, and drive type. Just as the `Car` class did, the `Truck` class calls the `Automobile` class's `__init__` method (in line 84) passing the make, model, mileage, and price as arguments. Line 87 creates the `__drive_type` attribute, initializing it to the value of the `drive_type` parameter.

The `set_drive_type` method in lines 92 through 93 is the mutator for the `__drive_type` attribute, and the `get_drive_type` method in lines 98 through 99 is the accessor for the attribute.

Now let's look at the `SUV` class, which also inherits from the `Automobile` class. The code for the `SUV` class, which is also in the `vehicles` module, is shown in Program 10-5.

Program 10-5 (Lines 101 through 128 of vehicles.py)

```
101 # The SUV class represents a sport utility vehicle. It
102 # is a subclass of the Automobile class.
103
104 class SUV(Automobile):
105     # The __init__ method accepts arguments for the
106     # SUV's make, model, mileage, price, and passenger
107     # capacity.
108
109     def __init__(self, make, model, mileage, price, pass-cap):
110         # Call the superclass's __init__ method and pass
111         # the required arguments. Note that we also have
112         # to pass self as an argument.
113         Automobile.--init--(self, make, model, mileage, price)
114
115         # Initialize the __pass_cap attribute.
116         self.--pass-cap = pass-cap
117
118     # The set-pass-cap method is the mutator for the
119     # __pass_cap attribute.
120
121     def set_pass_cap(self, pass-cap):
122         self.--pass_cap = pass-cap
123
124     # The get-pass-cap method is the accessor for the
125     # __pass_cap attribute.
126
127     def get_pass_cap(self):
128         return self.--pass-cap
```

The `SUV` class's `__init__` method begins in line 109. It takes arguments for the vehicle's make, model, mileage, price, and passenger capacity. Just as the `Car` and `Truck` classes did, the `SUV` class calls the `Automobile` class's `__init__` method (in line 113) passing the make, model, mileage, and price as arguments. Line 116 creates the `__pass_cap` attribute, initializing it to the value of the `pass-cap` parameter.

The `set_pass_cap` method in lines 121 through 122 is the mutator for the `__pass-cap` attribute, and the `get_pass_cap` method in lines 127 through 128 is the accessor for the attribute.

Program 10-6 demonstrates each of the classes we have discussed so far. It creates a `Car` object, a `Truck` object, and an `SUV` object.

Program 10-6 (car_truck_suv_demo.py)

```
# This program creates a Car object, a Truck object,
# and an SUV object.

import vehicles

def main():
    # Create a Car object for a used 2001 BMW
    # with 70,000 miles, priced at $15,000, with
    # 4 doors.
    car = vehicles.Car('BMW', 2001, 70000, 15000.0, 4)

    # Create a Truck object for a used 2002
    # Toyota pickup with 40,000 miles, priced
    # at $12,000, with 4-wheel drive.
    truck = vehicles.Truck('Toyota', 2002, 40000, 12000.0, '4WD')

    # Create an SUV object for a used 2000
    # Volvo with 30,000 miles, priced
    # at $18,500, with 5 passenger capacity.
    suv = vehicles.SUV('Volvo', 2000, 30000, 18500.0, 5)

    print 'USED CAR INVENTORY'
    print '==================='

    # Display the car's data.
    print 'The following car is in inventory:'
    print 'Make:', car.get_make()
    print 'Model:', car.get_model()
    print 'Mileage:', car.get_mileage()
    print 'Price:', car.get_price()
    print 'Number of doors:', car.get_doors()
    print

    # Display the truck's data.
    print 'The following pickup truck is in inventory.'
```

(program continues)

Program 10-6 *(continued)*

```
    print 'Make:', truck.get_make()
    print 'Model:', truck.get_model()
    print 'Mileage:', truck.get_mileage()
    print 'Price:', truck.get_price()
    print 'Drive type:', truck.get_drive_type()
    print

    # Display the SUV's data.
    print 'The following SUV is in inventory.'
    print 'Make:', suv.get_make()
    print 'Model:', suv.get_model()
    print 'Mileage:', suv.get_mileage()
    print 'Price:', suv.get_price()
    print 'Passenger Capacity:', suv.get_pass_cap()

# Call the main function.
main()
```

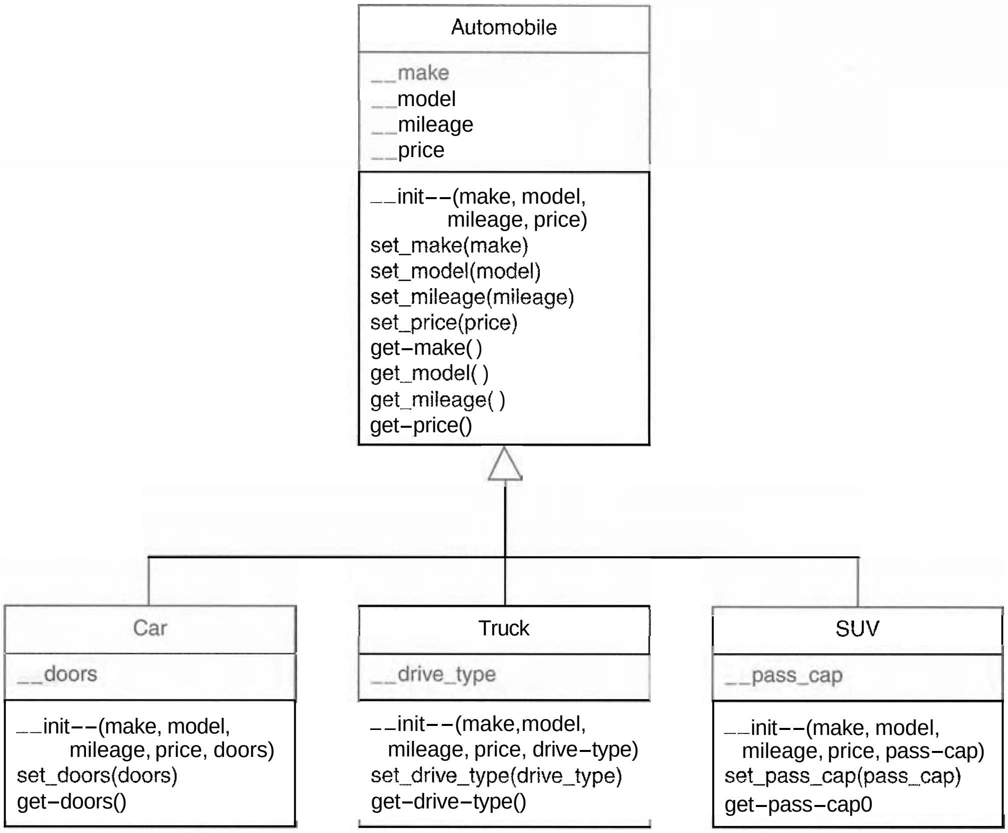
Program Output

```
USED CAR INVENTORY
==================
The following car is in inventory:
Make: BMW
Model: 2001
Mileage: 70000
Price: 15000.0
Number of doors: 4

The following pickup truck is in inventory.
Make: Toyota
Model: 2002
Mileage: 40000
Price: 12000.0
Drive type: 4WD

The following SUV is in inventory.
Make: Volvo
Model: 2000
Mileage: 30000
Price: 18500.0
Passenger Capacity: 5
```

Inheritance in UML Diagrams

You show inheritance in a UML diagram by drawing a line with an open arrowhead from the subclass to the superclass. (The arrowhead points to the superclass.) Figure 10-2 is a UML diagram showing the relationship between the `Automobile`, `Car`, `Truck`, and `SUV` classes.

Figure 10-2 UML diagram showing inheritance


Automobile
__make
__model
__mileage
__price
__init__(make, model, mileage, price)
set_make(make)
set_model(model)
set_mileage(mileage)
set_price(price)
get_make()
get_model()
get_mileage()
get_price()
Car
__doors
__init__(make, model, mileage, price, doors)
set_doors(doors)
get_doors()
Truck
__drive_type
__init__(make,model, mileage, price, drive_type)
set_drive_type(drive_type)
get_drive_type()
SUV
__pass_cap
__init__(make, model, mileage, price, pass_cap)
set_pass_cap(pass_cap)
get_pass_cap()

In the Spotlight:
Using Inheritance

Bank Financial Systems, Inc. develops financial software for banks and credit unions. The company is developing a new object-oriented system that manages customer accounts. One of your tasks is to develop a class that represents a savings account. The data that must be held by an object of this class is:

- The account number
- The interest rate
- The account balance

You must also develop a class that represents a certificate of deposit (CD) account. The data that must be held by an object of this class is:

- The account number
- The interest rate
- The account balance
- The account maturity date

As you analyze these requirements, you realize that a CD account is really a specialized version of a savings account. The class that represents a CD will hold all of the same data as the class that represents a savings account, plus an extra attribute for the maturity date. You decide to design a `SavingsAccount` class to represent a savings account, and then design a subclass of `SavingsAccount` named `CD` to represent a CD account. You will store both of these classes in a module named `accounts`. Program 10-7 shows the code for the `SavingsAccount` class.

Program 10-7 (Lines 1 through 37 of accounts.py)

```
# The SavingsAccount class represents a
# savings account.

class SavingsAccount:

    # The __init__ method accepts arguments for the
    # account number, interest rate, and balance.

    def __init__(self, account_num, int_rate, bal):
        self.__account_num = account_num
        self.__interest_rate = int_rate
        self.__balance = bal

    # The following methods are mutators for the
    # data attributes.

    def set_account_num(self, account_num):
        self.__account_num = account_num

    def set_interest_rate(self, int_rate):
        self.__interest_rate = int_rate

    def set_balance(self, bal):
        self.__balance = bal

    # The following methods are accessors for the
    # data attributes.

    def get_account_num(self):
        return self.__account_num

    def get_interest_rate(self):
        return self.__interest_rate

    def get_balance(self):
        return self.__balance

```

The class's `__init__` method appears in lines 9 through 12. The `__init__` method accepts arguments for the account number, interest rate, and balance. These arguments are used to initialize data attributes named `__account_num`, `__interest_rate`, and `__balance`.

The `set_account_num`, `set_interest_rate`, and `set_balance` methods that appear in lines 17 through 24 are mutators for the data attributes. The `get_account_num`, `get_interest_rate`, and `get_balance` methods that appear in lines 29 through 36 are accessors.

The CD class is shown in the next part of Program 10-7.

Program 10-7 (Lines 38 through 65 of accounts.py)

```
38  # The CD account represents a certificate of
39  # deposit (CD) account. It is a subclass of
40  # the SavingsAccount class.
41
42  class CD(SavingsAccount):
43
44      # The init method accepts arguments for the
45      # account number, interest rate, balance, and
46      # maturity date.
47
48      def __init__(self, account_num, int_rate, bal, mat_date):
49          # Call the superclass __init__ method.
50          SavingsAccount.__init__(self, account_num, int_rate, bal)
51
52          # Initialize the __maturity_date attribute.
53          self.__maturity_date = mat_date
54
55      # The set_maturity_date is a mutator for the
56      # __maturity_date attribute.
57
58      def set_maturity_date(self, mat_date):
59          self.__maturity_date = mat_date
60
61      # The get_maturity_date method is an accessor
62      # for the __maturity_date attribute.
63
64      def get_maturity_date(self):
65          return self.__maturity_date
```

The CD class's `__init__` method appears in lines 48 through 53. It accepts arguments for the account number, interest rate, balance, and maturity date. Line 50 calls the `SavingsAccount` class's `__init__` method, passing the arguments for the account number, interest rate, and balance. After the `SavingsAccount` class's `__init__` method executes, the `__account_num`, `__interest_rate`, and `__balance` attributes will be created and initialized. Then the statement in line 53 creates the `__maturity_date` attribute.

The `set_maturity_date` method in lines 58 through 59 is the mutator for the `__maturity_date` attribute, and the `get_maturity_date` method in lines 64 through 65 is the accessor.

To test the classes, we use the code shown in Program 10-8. This program creates an instance of the `SavingsAccount` class to represent a savings account, and an instance of the `CD` account to represent a certificate of deposit account.

Program 10-8 (account_demo.py)

```
# This program creates an instance of the SavingsAccount
# class and an instance of the CD account.

import accounts

def main():
    # Get the account number, interest rate,
    # and account balance for a savings account.
    print 'Enter the following data for a savings account.'
    acct_num = raw_input('Account number: ')
    int_rate = input('Interest rate: ')
    balance = input('Balance: ')

    # Create a SavingsAccount object.
    savings = accounts.SavingsAccount(acct_num, int_rate, balance)

    # Get the account number, interest rate,
    # account balance, and maturity date for a CD.
    print 'Enter the following data for a CD.'
    acct_num = raw_input('Account number: ')
    int_rate = input('Interest rate: ')
    balance = input('Balance: ')
    maturity = raw_input('Maturity date: ')

    # Create a SavingsAccount object.
    cd = accounts.CD(acct_num, int_rate, balance, maturity)

    # Display the data entered.
    print 'Here is the data you entered:'
    print
    print 'Savings Account'
    print '---------------'
    print 'Account number:', savings.get_account_num()
    print 'Interest rate:', savings.get_interest_rate()
    print 'Balance: $%.2f' % savings.get_balance()
    print
    print 'CD'
    print '---------------'
    print 'Account number:', cd.get_account_num()
    print 'Interest rate:', cd.get_interest_rate()
    print 'Balance: $%.2f' % cd.get_balance()
    print 'Maturity date:', cd.get_maturity_date()

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the following data for a savings account.
Account number: 1234SA [Enter]
Interest rate: 3.5 [Enter]
Balance: 1000.00 [Enter]
Enter the following data for a CD.
Account number: 2345CD [Enter]
Interest rate: 5.6 [Enter]
Balance: 2500.00 [Enter]
Maturity date: 12/12/2010 [Enter]
Here is the data you entered:

Savings Account
---------------
Account number: 1234SA
Interest rate: 3.5
Balance: $1000.00

CD
---------------
Account number: 2345CD
Interest rate: 5.6
Balance: $2500.00
Maturity date: 12/12/2010
```

Checkpoint

10.1 In this section we discussed superclasses and subclasses. Which is the general class and which is the specialized class?

10.2 What does it mean to say there is an "is a" relationship between two objects?

10.3 What does a subclass inherit from its superclass?

10.4 Look at the following code, which is the first line of a class definition. What is the name of the superclass? What is the name of the subclass?

```
class Canary(Bird):
```

10.2 Polymorphism

CONCEPT: Polymorphism allows subclasses to have methods with the same names as methods in their superclasses. it gives the ability for a program to call the correct method depending on the type of object that is used to call it.

The term *polymorphism* refers to an object's ability to take different forms. It is a powerful feature of object-oriented programming. In this section, we will look at two essential ingredients of polymorphic behavior:

1. The ability to define a method in a superclass, and then define a method with the same name in a subclass. When a subclass method has the same name as a superclass method, it is often said that the subclass method *overrides* the superclass method.
2. The ability to call the correct version of an overridden method, depending on the type of object that is used to call it. If a subclass object is used to call an overridden method, then the subclass's version of the method is the one that will execute. If a superclass object is used to call an overridden method, then the superclass's version of the method is the one that will execute.

Actually, you've already seen method overriding at work. Each subclass that we have examined in this chapter has a method named `__init__` that overrides the superclass's `__init__` method. When an instance of the subclass is created, it is the subclass's `__init__` method that automatically gets called.

Method overriding works for other class methods too. Perhaps the best way to describe polymorphism is to demonstrate it, so let's look at a simple example. Program 10-9 shows the code for a class named `Mammal`, which is in a module named `animals`.

Program 10-9 (Lines 1 through 22 of animals.py)

```
# The Mammal class represents a generic mammal.

class Mammal:

    # The __init__ method accepts an argument for
    # the mammal's species.

    def __init__(self, species):
        self.__species = species

    # The show_species method displays a message
    # indicating the mammal's species.

    def show_species(self):
        print 'I am a', self.__species

    # The make_sound method is the mammal's
    # way of making a generic sound.

    def make_sound(self):
        print 'Grrrrr'

```

The `Mammal` class has three methods: `__init__`, `show—species` and `make—sound`. Here is an example of code that creates an instance of the class and calls the uses these methods:

```
import animals
mammal = animals.Mammal('regular mammal')
mammal.show_species()
mammal.make_sound()
```

This code will display the following:

```
I am a regular mammal
Grrrrr
```

The next part of Program 10-9 shows the `Dog` class. The `Dog` class, which is also in the `animals` module, is a subclass of the `Mammal` class.

Program 10-9 (Lines 23 through 38 of animals.py)

```
# The Dog class is a subclass of the Mammal class.

class Dog(Mammal):

    # The __init__ method calls the superclass's
    # __init__ method passing 'Dog' as the species.

    def __init__(self):
        Mammal.-init--(self, 'Dog')

    # The make-sound method overrides the superclass's
    # make-sound method.

    def make_sound(self):
        print 'Woof! Woof!'

```

Even though the Dog class inherits the `__init__` and `make_sound` methods that are in the `Mammal` class, those methods are not adequate for the `Dog` class. So, the `Dog` class has its own `__init__` and `make_sound` methods, which perform actions that are more appropriate for a dog. We say that the `__init__` and `make_sound` methods in the `Dog` class override the `__init__` and `make_sound` methods in the Mammal class. Here is an example of code that creates an instance of the `Dog` class and calls the methods:

```
import animals
dog = animals.Dog()
```

```
dog.show_species()
dog.make_sound()
```

This code will display the following:

```
I am a Dog
Woof! Woof!
```

When we use a `Dog` object to call the `show_species` and make—sound methods, the versions of these methods that are in the `Dog` class are the ones that execute. Next, look at Program 10-10, which shows the `Cat` class. The `Cat` class, which is also in the `animals` module, is another subclass of the `Mammal` class.

Program 10-9 (Lines 39 through 53 of animals.py)

```
# The Cat class is a subclass of the Mammal class.

class Cat(Mammal):

    # The __init__ method calls the superclass's
    # __init__ method passing 'Cat' as the species.

    def __init__(self):
        Mammal.--init--(self, 'Cat')

    # The make-sound method overrides the superclass's
    # make-sound method.

    def make_sound(self):
        print 'Meow'
```

The `Cat` class also overrides the Mammal class's `__init__` and make_sound methods. Here is an example of code that creates an instance of the `Cat` class and calls these methods:

```
import animals
cat = animals.Dog()
cat.show_species()
cat.make_sound()
```

This code will display the following:

```
I am a Cat
Meow
```

When we use a `Cat` object to call the `show_species` and make—sound methods, the versions of these methods that are in the `Cat` class are the ones that execute.

The `isinstance` Function

Polymorphism gives us a great deal of flexibility when designing programs. For example, look at the following function:

```
def show_mammal_info(creature):
    creature.show_species()
    creature.make-sound()
```

We can pass any object as an argument to this function, and as long as it has a `show_species` method and a `make_sound` method, the function will call those methods. In essence, we can pass any object that "is a" `Mammal` (or a subclass of `Mammal`) to the function. Program 10-10 demonstrates.

Program 10-10 (polymorphism_demo.py)

```
# This program demonstrates polymorphism.

import animals

def main():
    # Create a Mammal object, a Dog object, and
    # a Cat object.
    mammal = animals.Mammal('regular animal')
    dog = animals.Dog( )
    cat = animals.Cat( )

    # Display information about each one.
    print 'Here are some animals and'
    print 'the sounds they make.'
    print '-------------------------'
    show_mammal_info(mammal)
    print
    show_mammal_info(dog)
    print
    show_mammal_info(cat)

# The show-mammal-info function accepts an object
# as an argument, and calls its show-species
# and make-sound methods.

def show_mammal_info(creature):
    creature.show_species()
    creature.make-sound()

# Call the main function.
main()
```

Program Output

```
Here are some animals and
the sounds they make.
--------------------------
I am a regular animal
Grrrrr

I am a Dog
Woof! Woof!

I am a Cat
Meow
```

But what happens if we pass an object that is not a `Mammal`, and not of a subclass of `Mammal` to the function? For example, what will happen when Program 10-11 runs?

Program 10-11 (wrong_type.py)

```
 1  def main():
 2      # Pass a string to show-mammal-info...
 3      show_mammal_info('I am a string')
 4
 5  # The show-mammal-info function accepts an object
 6  # as an argument, and calls its show-species
 7  # and make-sound methods.
 8
 9  def show_mammal_info(creature):
10      creature.show_species()
11      creature·make-sound()
12
13  # Call the main function.
14  main()
```

In line 3 we call the show_mammal—info function passing a string as an argument. When the interpreter attempts to execute line 10, however, an `AttributeError` exception will be raised because strings do not have a method named show_species.

We can prevent this exception from occurring, by using the built-in function isinstance. You can use the isinstance function to determine whether an object is an instance of a specific class, or a subclass of that class. Here is the general format of the function call:

`isinstance(`*`object`*`, ClassName)`

In the general format, *`object`* is a reference to an object and *`ClassName`* is the name of a class. If the object referenced by *`object`* is an instance of *`ClassName`* or is an instance of a subclass of *`ClassName`*, the function returns true. Otherwise it returns false. Program 10-12 shows how we can use it in the show—mammal_info function.

Program 10-12 (polymorphism_demo2.py)

```
# This program demonstrates polymorphism.

import animals

def main():
    # Create an Mammal object, a Dog object, and
    # a Cat object.
    mammal = animals.Mammal('regular animal')
    dog = animals.Dog()
    cat = animals.Cat()

    # Display information about each one.
    print 'Here axe some animals and'
    print 'the sounds they make.'
    print '--------------------------'
    show_mammal_info(mammal)
    print
    show_mammal_info(dog)
    print
    show_mammal_info(cat)
    print
    show_mammal_info('I am a string')

# The show-mammal-info function accepts an object
# as an argument, and calls its show-species
# and make-sound methods.

def show_mammal_info(creature):
    if isinstance(creature, animals.Mammal):
        creature.show_species()
        creature.make_sound()
    else:
        print 'That is not a Mammal!'

# Call the main function.
main()
```

Program Output

```
Here are some animals and
the sounds they make.
--------------------------
I am a regular animal
Grrrrr
```

(program output continues)

Program Output *(continued)*

```
I am a Dog
Woof! Woof!

I am a Cat
Meow

That is not a Mammal!
```

In lines 16, 18, and 20 we call the `show_mammal_info` function, passing references to a `Mammal` object, a `Dog` object, and a `Cat` object. In line 22, however, we call the function and pass a string as an argument. Inside the `show_mammal_info` function, the `if` statement in line 29 calls the `isinstance` function to determine whether the argument is an instance of `Mammal` (or a subclass). If it is not, an error message is displayed.

Checkpoint

6.25 Look at the following class definitions:

```
class Vegetable:
    def __init--(self, vegtype):
        self.__vegtype = vegtype

    def message(self):
        print "I'm a vegetable."

class Potato(Vegetable):
    def __init__(self):
        Vegetable.--init--(self, 'potato')

    def message(self):
        print "I'm a potato."
```

Given these class definitions, what will the following statements display?

```
v = Vegetable('veggie')
p = Potato()
v.message()
p.message()
```

Review Questions

Multiple Choice

1. In an inheritance relationship, the ________ is the general class.
   a. subclass
   b. superclass
   c. slave class
   d. child class

2. In an inheritance relationship, the ________ is the specialized class.
   a. superclass
   b. master class
   c. subclass
   d. parent class
3. Suppose a program uses two classes: `Airplane` and `JumboJet`. Which of these would most likely be the subclass?
   a. `Airplane`
   b. `JumboJet`
   c. Both
   d. Neither
4. This characteristic of object-oriented programming allows the correct version of an overridden method to be called when an instance of a subclass is used to call it.
   a. polymorphism
   b. inheritance
   c. generalization
   d. specialization
5. You can use this to determine whether an object is an instance of a class.
   a. The `in` operator
   b. The `is_object_of` function
   c. The `isinstance` function
   d. The error messages that are displayed when a program crashes

True or False

1. Polymorphism allows you to write methods in a subclass that have the same name as methods in the superclass.
2. It is not possible to call a superclass's `__init__` method from a subclass's `__init__` method.
3. A subclass can have a method with the same name as a method in the superclass.
4. Only the `__init__` method can be overridden.
5. You cannot use the `isinstance` function to determine whether an object is an instance of a subclass of a class.

Short Answer

1. What does a subclass inherit from its superclass?
2. Look at the following class definition. What is the name of the superclass? What is the name of the subclass?

   ```
   class Tiger(Felis):
   ```
3. What is an overridden method?

Algorithm Workbench

1. Write the first line of the definition for a `Poodle` class. The class should extend the `Dog` class.

2. Look at the following class definitions:

```
class Plant:
    def __init__(self, plant_type):
        self.__plant_type = plant_type

    def message(self):
        print "I'm a plant."

class Tree(Plant):
    def __init__(self):
        Plant.__init__(self, 'tree')

    def message(self):
        print "I'm a tree."
```

Given these class definitions, what will the following statements display?

```
p = Plant('sapling')
t = Tree()
p.message()
t.message()
```

3. Look at the following class definition:

```
class Beverage:
    def __init__(self, bev_name):
        self.__bev_name = bev_name
```

Write the code for a class named `Cola` that is a subclass of the `Beverage` class. The `Cola` class's `__init__` method should call the `Beverage` class's `__init__` method, passing 'cola' as an argument.

Programming Exercises

1. Employee and ProductionWorker Classes

Write an `Employee` class that keeps data attributes for the following pieces of information:

- Employee name
- Employee number

Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data attributes for the following information:

- Shift number (an integer, such as 1, 2, or 3)
- Hourly pay rate

The workday is divided into two shifts: day and night. The shift attribute will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate accessor and mutator methods for each class.

Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data attributes. Store the data in the object and then use the object's accessor methods to retrieve it and display it on the screen.

2. `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in Programming Exercise 1. The `ShiftSupervisor` class should keep a data attribute for the annual salary and a data attribute for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. `Person` and `Customer` Classes

Write a class named `Person` with data attributes for a person's name, address, and telephone number. Next, write a class named `Customer` that is a subclass of the `Person` class. The `Customer` class should have a data attribute for a customer number and a Boolean data attribute indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.

CHAPTER 11 Recursion

TOPICS

11.1 Introduction to Recursion
11.2 Problem Solving with Recursion
11.3 Examples of Recursive Algorithms

11.1 Introduction to Recursion

CONCEPT: A recursive function is a function that calls itself.

You have seen instances of functions calling other functions. In a program, the `main` function might call function `A`, which then might call function `B`. It's also possible for a function to call itself. A function that calls itself is known as a *recursive function*. For example, look at the `message` function shown in Program 11-1.

Program 11-1 (endless_recursion.py)

```
# This program has a recursive function.

def main():
    message()

def message():
    print 'This is a recursive function.'
    message()

# Call the main function.
main()
```

Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

. . . and this output repeats forever!

The **`message`** function displays the string 'This is a recursive function' and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating.

Like a loop, a recursive function must have some way to control the number of times it repeats. The code in Program 11-2 shows a modified version of the **`message`** function. In this program, the **`message`** function receives an argument that specifies the number of times the function should display the message.

Program 11-2

```
 1  # This program has a recursive function.
 2
 3  def main():
 4      # By passing the argument 5 to the message
 5      # function we are telling it to display the
 6      # message five times.
 7      message(5)
 8
 9  def message(times):
10      if (times > 0):
11          print 'This is a recursive function.'
12          message(times - 1)
13
14  # Call the main function.
15  main()
```

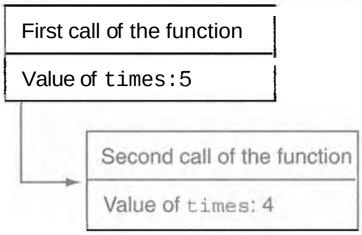
Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

The **`message`** function in this program contains an `if` statement in line 10 that controls the repetition. As long as the **`times`** parameter is greater than zero, the message 'This is a recursive function' is displayed, and then the function calls itself again, but with a smaller argument.

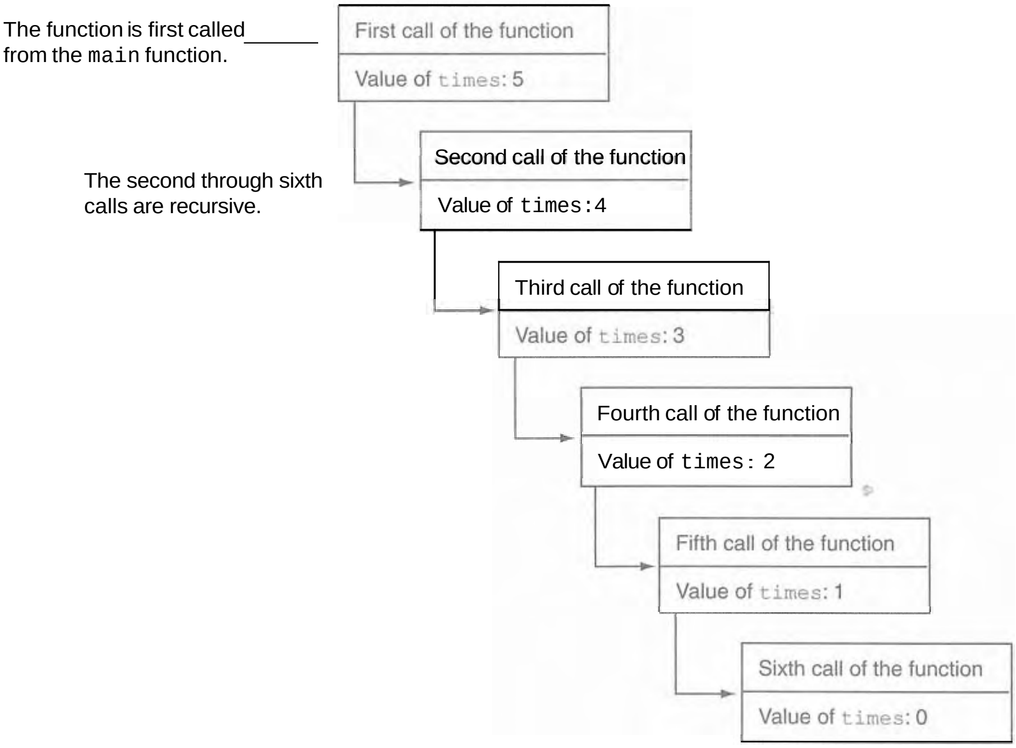
In line 7 the `main` function calls the `message` function passing the argument 5. The first time the function is called the `if` statement displays the message and then calls itself with 4 as the argument. Figure 11-1 illustrates this.

Figure 11-1 First two calls of the function


First call of the function
Value of times:5
Second call of the function
Value of times: 4

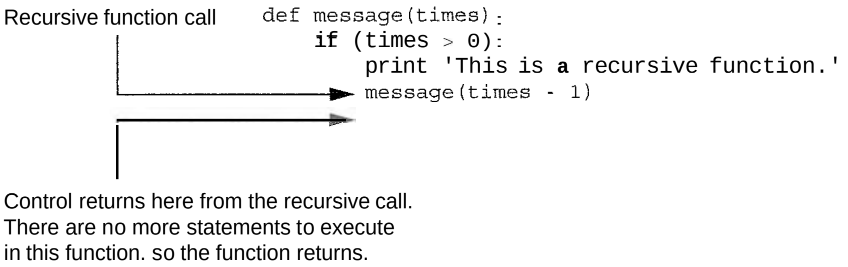
The diagram shown in Figure 11-1 illustrates two separate calls of the `message` function. Each time the function is called, a new instance of the `times` parameter is created in memory. The first time the function is called, the `times` parameter is set to 5. When the function calls itself, a new instance of the `times` parameter is created, and the value 4 is passed into it. This cycle repeats until finally, zero is passed as an argument to the function. This is illustrated in Figure 11-2.

Figure 11-2 Six calls to the `message` function


The function is first called from the main function.
The second through sixth calls are recursive.
First call of the function
Value of times: 5
Second call of the function
Value of times:4
Third call of the function
Value of times: 3
Fourth call of the function
Value of times: 2
Fifth call of the function
Value of times: 1
Sixth call of the function
Value of times: 0

As you can see in the figure, the function is called six times. The first time it is called from the `main` function, and the other five times it calls itself. The number of times that a function calls itself is known as the *depth* of *recursion.* In this example, the depth of recursion is five. When the function reaches its sixth call, the `times` parameter is set to 0. At that point, the `if` statement's conditional expression is false, so the function returns. Control of the program returns from the sixth instance of the function to the point in the fifth instance directly after the recursive function call. This is illustrated in Figure 11-3.

Figure 11-3 Control returns to the point after the recursive function call



Recursive function call

```
def message(times):
    if (times > 0):
        print 'This is a recursive function.'
        message(times - 1)
```

Control returns here from the recursive call.
There are no more statements to execute
in this function. so the function returns.

Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return.

11.2 Problem Solving with Recursion

— CONCEPT: **A** problem can be solved with recursion if it can be broken down into smaller problems that are identical in structure to the overall problem.

The code shown in Program 11-2 demonstrates the mechanics of a recursive function. Recursion can be a powerful tool for solving repetitive problems and is commonly studied in upper-level computer science courses. It may not yet be clear to you how to use recursion to solve a problem.

First, note that recursion is never required to solve a problem. Any problem that can be solved recursively can also be solved with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because the process of calling a function requires several actions to be performed by the computer. These actions include allocating memory for parameters and local variables, and storing the address of the program location where control returns after the function terminates. These actions, which are sometimes referred to as *overhead,* take place with each function call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with a loop. Where a loop might result in faster execution time, the programmer might be able to design a recursive algorithm faster. In general, a recursive function works as follows:

- If the problem can be solved now, without recursion, then the function solves it and returns
- If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem

In order to apply this approach, first, we identify at least one case in which the problem can be solved without recursion. This is known as the *base case.* Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case.* In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

Using Recursion to Calculate the Factorial of a Number

Let's take an example from mathematics to examine an application of recursive functions. In mathematics, the notation n! represents the factorial of the number n. The factorial of a nonnegative number can be defined by the following rules:

If $n = 0$ then $n! = 1$
If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$

Let's replace the notation n! with factorial(n), which looks a bit more like computer code, and rewrite these rules as follows:

If $n = 0$ then $\text{factorial}(n) = 1$
If $n > 0$ then $\text{factorial}(n) = 1 \times 2 \times 3 \times \ldots \times n$

These rules state that when n is 0, its factorial is 1. When n is greater than 0, its factorial is the product of all the positive integers from 1 up to n. For instance, factorial(6) is calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$.

When designing a recursive algorithm to calculate the factorial of any number, first we identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where n is equal to 0 as follows:

If $n = 0$ then $\text{factorial}(n) = 1$

This tells how to solve the problem when n is equal to 0, but what do we do when n is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it: 3

If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$

This states that if n is greater than 0, the factorial of n is n times the factorial of $n - 1$. Notice how the recursive call works on a reduced version of the problem, $n - 1$. So, our recursive rule for calculating the factorial of a number might look like this:

If $n = 0$ then $\text{factorial}(n) = 1$
If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$

The code in Program 11-3 shows how we might design a factorial function in a program.

Program 11-3

```
# This program uses recursion to calculate
# the factorial of a number.

def main():
    # Get a number from the user.
    number = input('Enter a nonnegative integer: ')

    # Get the factorial of the number.
    fact = factorial(number)

    # Display the factorial.
    print 'The factorial of', number, 'is', fact

# The factorial function uses recursion to
# calculate the factorial of its argument,
# which is assumed to be nonnegative.
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)

# Call the main function.
main()
```

Program Output (with input shown in bold)

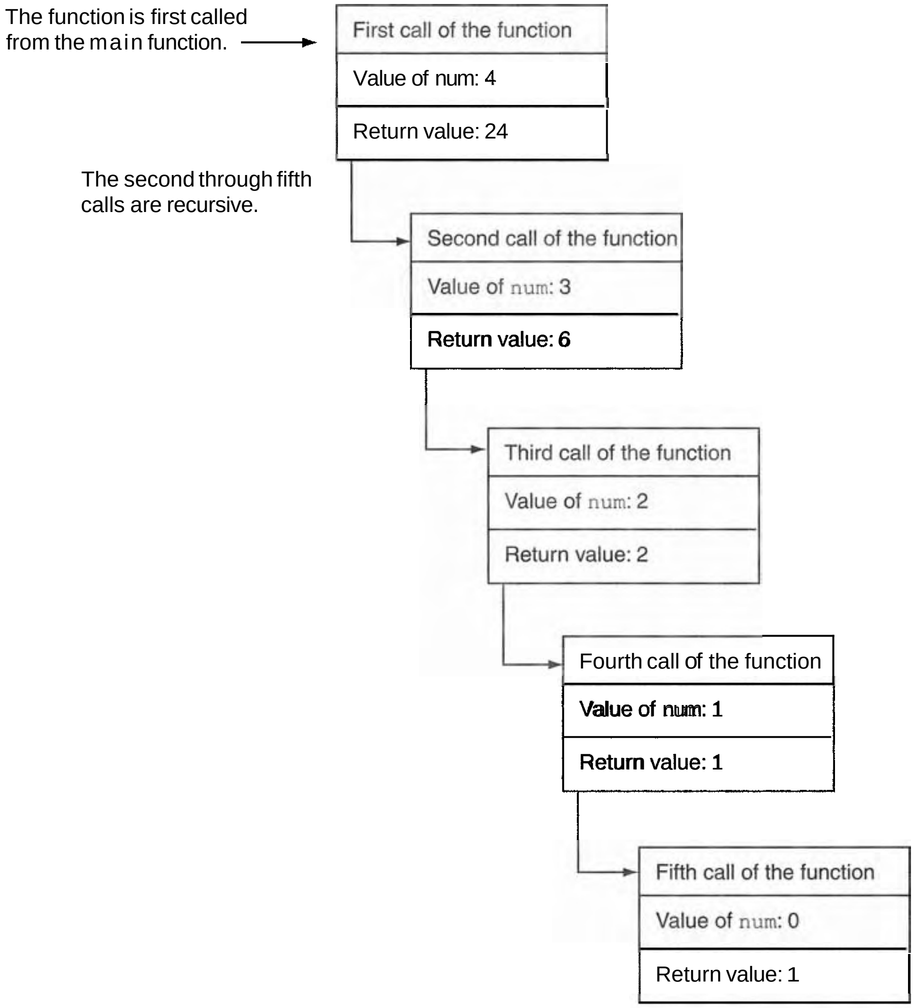
```
Enter a nonnegative integer: 4 [Enter]
The factorial of 4 is 24
```

In the sample run of the program, the `factorial` function is called with the argument 4 passed to num. Because num is not equal to 0, the `if` statement's `else` clause executes the following statement:

```
return num * factorial(num - 1)
```

Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(num - 1)` must be determined. The `factorial` function is called recursively until the fifth call, in which the num parameter will be set to zero. Figure 11-4 illustrates the value of num and the return value during each call of the function.

Figure 11-4 The value of num and the return value during each call of the function


The function is first called from the main function.
First call of the function
Value of num: 4
Return value: 24
The second through fifth calls are recursive.
Second call of the function
Value of num: 3
Return value: 6
Third call of the function
Value of num: 2
Return value: 2
Fourth call of the function
Value of num: 1
Return value: 1
Fifth call of the function
Value of num: 0
Return value: 1

The figure illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually, the recursion has to stop in order for a solution to be reached.

If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our **`factorial`** function, the value of the parameter `num` gets closer to 0 with each recursive call. When the parameter reaches 0, the function returns a value without making another recursive call.

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions or functions that directly call themselves. This is known as *direct recursion.* There is also the possibility of creating indirect recursion in a program. This occurs when function `A` calls function `B`, which in turn calls function `A`. There can even be several functions involved in the recursion. For example, function `A` could call function `B`, which could call function `C`, which calls function `A`.

Checkpoint

11.1 It is said that a recursive algorithm has more overhead than an iterative algorithm. What does this mean?

11.2 What is a base case?

11.3 What is a recursive case?

11.4 What causes a recursive algorithm to stop calling itself?

11.5 What is direct recursion? What is indirect recursion?

11.3 Examples of Recursive Algorithms

Summing a Range of List Elements with Recursion

In this example, we look at a function named `range_sum` that uses recursion to sum a range of items in a list. The function takes the following arguments: a list that contains the range of elements to be summed, an integer specifying the index of the starting item in the range, and an integer specifying the index of the ending item in the range. Here is an example of how the function might be used:

```
numbers= [1, 2, 3, 4, 5, 6, 7, 8, 9]
my-sum = range_sum(numbers, 3, 7)
print my-sum
```

The second statement in this code specifies that the `range_sum` function should return the sum of the items at indexes 3 through 7 in the `numbers` list. The return value, which in this case would be 30, is assigned to the `my_sum` variable. Here is the definition of the `range_sum` function:

```
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)
```

This function's base case is when the `start` parameter is greater than the `end` parameter. If this is true, the function returns the value 0. Otherwise, the function executes the following statement:

```
return num_list[start] + range_sum(num_list, start + 1, end)
```

This statement returns the sum of `num_list[start]` plus the return value of a recursive call. Notice that in the recursive call, the starting item in the range is `start + 1`. In essence, this statement says "return the value of the first item in the range plus the sum of the rest of the items in the range." Program 11-4 demonstrates the function.

Program 11-4

```
# This program demonstrates the range-sum function.

def main():
    # Create a list of numbers.
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

    # Get the sum of the items at indexes 2
    # through 5.
    my_sum = range_sum(numbers, 2, 5)

    # Display the sum.
    print 'The sum of items 2 through 5 is', my-sum

# The range-sum function returns the sum of a specified
# range of items in num_list. The start parameter
# specifies the index of the starting item. The end
# parameter specifies the index of the ending item.
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)

# Call the main function.
main()
```

Program Output

```
The sum of elements 2 through 5 is 18
```

The Fibonacci Series

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of Fibonacci numbers. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . .

Notice that after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as follows:

If $n = 0$ then $\quad \text{Fib}(n) = 0$
If $n = 1$ then $\quad \text{Fib}(n) = 1$
If $n > 1$ then $\quad \text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

A recursive function to calculate the nth number in the Fibonacci series is shown here:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Notice that this function actually has two base cases: when n is equal to 0, and when *n* is equal to 1. In either case, the function returns a value without making a recursive call. The code in Program 11-5 demonstrates this function by displaying the first 10 numbers in the Fibonacci series.

Program 11-5 (fibonacci.py)

```
# This program uses recursion to print numbers
# from the Fibonacci series.

def main():
    print 'The first 10 numbers in the'
    print 'Fibonacci series are:'

    for number in range(1, 11):
        print fib(number)

# The fib function returns the nth number
# in the Fibonacci series.
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

# Call the main function.
main()
```

Program Output

```
The first 10 numbers in the
Fibonacci series are:
0
```

```
1
1
2
3
5
8
13
21
34
```

Finding the Greatest Common Divisor

Our next example of recursion is the calculation of the greatest common divisor (GCD) of two numbers. The GCD of two positive integers x and y is determined as follows:

> If x can be evenly divided by y, then gcd(x, y) = y
> Otherwise, gcd(x, y) = gcd(y, remainder of x/y)

This definition states that the GCD of x and y is y if x/y has no remainder. This is the base case. Otherwise, the answer is the GCD of y and the remainder of x/y. The code in Program 11-6 shows a recursive method for calculating the GCD.

Program 11-6 (gcd.py)

```
# This program uses recursion to find the GCD
# of two numbers.

def main():
    # Get two numbers.
    numl = input('Enter an integer: ')
    num2 = input('Enter another integer: ')

    # Display the GCD.
    print 'The greatest common divisor of'
    print 'the two numbers is', gcd(numl, num2)

# The gcd function returns the greatest common
# divisor of two numbers.
def gcd(x, y):
    if x % y == 0:
        return y
    else:
        return gcd(x, x % y)

# Call the main function.
main()
```

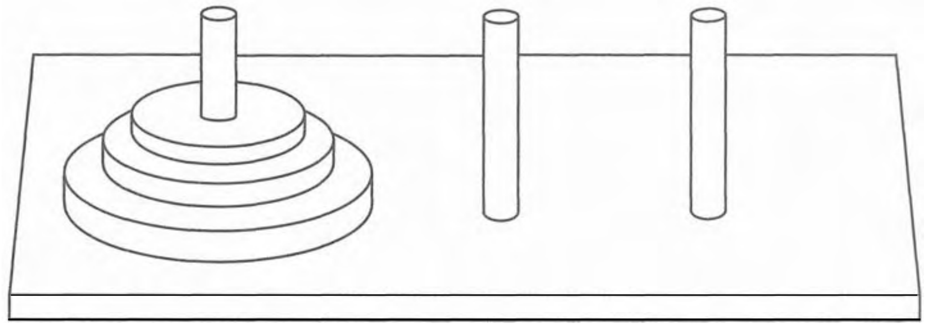
Program Output (with input shown in bold)

```
Enter an integer: 49 [Enter]
Enter another integer: 28 [Enter
The greatest common divisor of
these two numbers is 7
```

The Towers of Hanoi

The Towers of Hanoi is a mathematical game that is often used in computer science to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 11-5.

Figure 11-5 The pegs and discs in the Tower of Hanoi game



Notice that the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend where a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disk may be moved at a time
- A disk cannot be placed on top of a smaller disc
- All discs must be stored on a peg except while being moved

According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.[1]

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you only have one disc, the solution to the game is simple: move the disc from peg 1 to peg **3.** If you have two discs, the solution requires three moves:

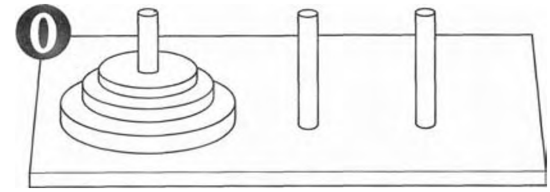
---

[1] In case you're worried about the monks finishing their job and causing the world to end anytime soon, you can relax. If the monks move the discs at a rate of 1 per second, it will take them approximately 585 billion years to move all 64 discs!

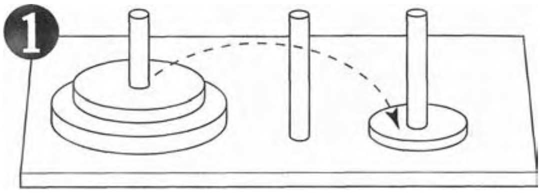
- Move disc **1** to peg 2
- Move disc 2 to peg **3**
- Move disc **1** to peg **3**

Notice that this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increases. To move three discs requires the seven moves shown in Figure 11-6.

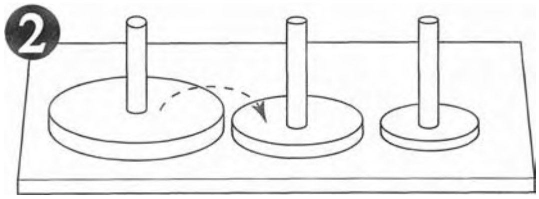
Figure 11-6 Steps for moving three pegs



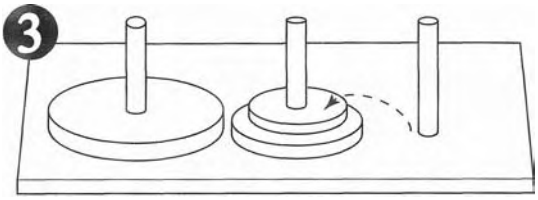
Original setup.



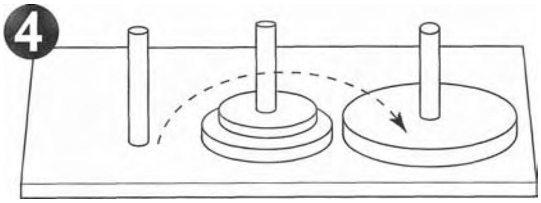
First move: Move disc 1 to peg 3.



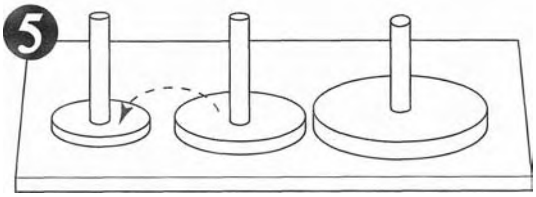
Second move: Move disc 2 to peg 2.



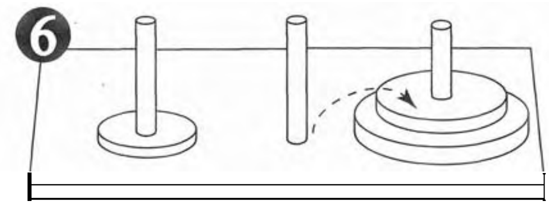
Third move: Move disc 1 to peg 2.



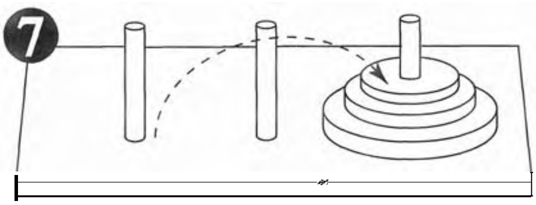
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

The following statement describes the overall solution to the problem:

> Move n discs from peg 1 to peg **3** using peg 2 as a temporary peg.

The following summary describes a recursive algorithm that simulates the solution to the game. Notice that in this algorithm we use the variables A, B, and C to hold peg numbers.

```
To move n discs from peg A to peg C, using peg B as a temporary peg, do the following.
If n > 0:
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.
    Move the remaining disc from peg A to peg C.
    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.
```

The base case for the algorithm is reached when there are no more discs to move. The following code is for a function that implements this algorithm. Note that the function doe: not actually move anything, but displays instructions indicating all of the disc moves tc make.

```
def move_discs(num, from-peg, to_peg, temp-peg):
    if num > 0:
        move_discs(num - 1, from-peg, temp-peg, to-peg)
        print 'Move a disc from peg', from-peg, 'to peg', to-peg
        move_discs(num - 1, temp-peg, to-peg, from-peg)
```

This function accepts arguments into the following parameters:

num	The number of discs to move.
`from_peg`	The peg to move the discs from.
`to_peg`	The peg to move the discs to.
`temp_peg`	The peg to use as a temporary peg.

If num is greater than 0, then there are discs to move. The first recursive call is as follows:

```
move_discs(num - 1, from-peg, temp-peg, to-peg)
```

This statement is an instruction to move all but one disc from from—peg to temp—peg, using to—peg as a temporary peg. The next statement is as follows:

```
print 'Move a disc from peg', from-peg, 'to peg', to_peg
```

This simply displays a message indicating that a disc should be moved from `from_peg` to `to—peg`. Next, another recursive call is executed as follows:

```
move-discs(num - 1, temp_peg, to-peg, from-peg)
```

This statement is an instruction to move all but one disc from temp—peg to to—peg, using `from_peg` as a temporary peg. The code in Program 11-7 demonstrates the function by displaying a solution for the Tower of Hanoi game.

Program 11-7 (towers_of_hanoi.py)

```
# This program simulates the Towers of Hanoi game.

def main():
    # Set up some initial values.
    num_discs = 3
    from_peg = 1
    to_peg = 3
    temp-peg =2

    # Play the game.
    move_discs(num_discs, from-peg, to-peg, temp-peg)
    print 'All the pegs are moved!'

# The moveDiscs function displays a disc move in
# the Towers of Hanoi game.
# The parameters are:
#    num:      The number of discs to move.
#    from-peg: The peg to move from.
#    to_peg:   The peg to move to.
#    temp-peg: The temporary peg.
def move_discs(num, from-peg, to-peg, temp-peg):
    if num > 0:
        move_discs(num - 1, from-peg, temp-peg, to-peg)
        print 'Move a disc from peg', from-peg, 'to peg', to-peg
        move_discs(num - 1, temp-peg, to-peg, from-peg)

# Call the main function.
main()
```

Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

Recursion versus Looping

Any algorithm that can be coded with recursion can also be coded with a loop. Both approaches achieve repetition, but which is best to use?

There are several reasons not to use recursion. Recursive function calls are certainly less efficient than loops. Each time a function is called, the system incurs overhead that is not necessary with a loop. Also, in many cases, a solution using a loop is more evident than a recursive solution. In fact, the majority of repetitive programming tasks are best done with loops.

Some problems, however, are more easily solved with recursion than with a loop. For example, the mathematical definition of the GCD formula is well suited to a recursive approach. If a recursive solution is evident for a particular problem, and the recursive algorithm does not slow system performance an intolerable amount, then recursion would be a good design choice. If a problem is more easily solved with a loop, however, you should take that approach.

Review Questions

Multiple Choice

1. A recursive function ______________.
   a. calls a different function
   b. abnormally halts the program
   c. calls itself
   d. can only be called once
2. A function is called once from a program's `main` function, and then it calls itself four times. The depth of recursion is ______________.
   a. one
   b. four
   c. five
   d. nine
3. The part of a problem that can be solved without recursion is the ______________ case.
   a. base
   b. solvable
   c. known
   d. iterative
4. The part of a problem that is solved with recursion is the ______________ case.
   a. base
   b. iterative
   c. unknown
   d. recursion
5. When a function explicitly calls itself it is called ______________ recursion.
   a. explicit
   b. modal
   c. direct
   d. indirect
6. When function A calls function B, which calls function A it is called ______________ recursion.
   a. implicit
   b. modal
   c. direct
   d. indirect
7. Any problem that can be solved recursively can also be solved with a ______________.
   a. decision structure
   b. loop
   c. sequence structure
   d. case structure
8. Actions taken by the computer when a function is called, such as allocating memory for parameters and local variables, are referred to as ______________.
   a. overhead
   b. set up

   c. clean up
   d. synchronization
9. A recursive algorithm must ______________ in the recursive case.
   a. solve the problem without recursion
   b. reduce the problem to a smaller version of the original problem
   c. acknowledge that an error has occurred and abort the program
   d. enlarge the problem to a larger version of the original problem
10. A recursive algorithm must ______________ in the base case.
   a. solve the problem without recursion
   b. reduce the problem to a smaller version of the original problem
   c. acknowledge that an error has occurred and abort the program
   d. enlarge the problem to a larger version of the original problem

True or False

1. An algorithm that uses a loop will usually run faster than an equivalent recursive algorithm.
2. Some problems can be solved through recursion only.
3. It is not necessary to have a base case in all recursive algorithms.
4. In the base case, a recursive method calls itself with a smaller version of the original problem.

Short Answer

1. In Program 11-2, presented earlier in this chapter, what is the base case of the message function?
2. In this chapter, the rules given for calculating the factorial of a number are as follows:

   If $n = 0$ then factorial(n) = 1
   If $n > 0$ then factorial(n) = $n \times$ factorial($n - 1$)

   If you were designing a function from these rules, what would the base case be? What would the recursive case be?
3. Is recursion ever required to solve a problem? What other approach can you use to solve a problem that is repetitive in nature?
4. When recursion is used to solve a problem, why must the recursive function call itself to solve a smaller version of the original problem?
5. How is a problem usually reduced with a recursive function?

Algorithm Workbench

1. What will the following program display?

```
def main():
    num = 0
    show_me(num)

def show_me(arg):
    if arg < 10:
        show_me(arg + 1)
    else:
        print arg

main()
```

2. What will the following program display?

```
def main():
    num = 0
    show_me(num)

def show_me(arg):
    print arg
    if arg  10:
        show_me(arg + 1)
main()
```

3. The following function uses a loop. Rewrite it as a recursive function that performs the same operation.

```
def trafficSign(n):
    while n > 0:
        print ‘No Parking’
        n = n – 1
```

Programming Exercises

1. Recursive Printing

Design a recursive function that accepts an integer argument, n, and prints the numbers 1 up through n.

2. Recursive Multiplication

Design a recursive function that accepts two arguments into the parameters `x` and `y`. The function should return the value of `x` times `y`. Remember, multiplication can be performed as repeated addition as follows:

$$7 \times 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4$$

(To keep the function simple, assume that `x` and `y` will always hold positive nonzero integers.)

3. Recursive Lines

Write a recursive function that accepts an integer argument, `n`. The function should display `n` lines of asterisks on the screen, with the first line showing 1 asterisk, the second line showing 2 asterisks, up to the nth line which shows `n` asterisks.

4. Largest List Item

Design a function that accepts a list as an argument, and returns the largest value in the list. The function should use recursion to find the largest item.

5. Recursive List Sum

Design a function that accepts a list of numbers as an argument. The function should recursively calculate the sum of all the numbers in the list and return that value.

6. Sum of Numbers

Design a function that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the function will return the sum of 1, 2, 3, 4, . . . 50. Use recursion to calculate the sum.

7. Recursive Power Method

Design a function that uses recursion to raise a number to a power. The function should accept two arguments: the number to be raised and the exponent. Assume that the exponent is a nonnegative integer.

8. Ackermann's Function

Ackermann's Function is a recursive mathematical algorithm that can be used to test how well a system optimizes its performance of recursion. Design a function `ackermann(m, n)`, which solves Ackermann's function. Use the following logic in your function:

```
If $m = 0$ then return $n + 1$
If $n = 0$ then return ackermann($m - 1$, 1)
Otherwise, return ackermann($m - 1$, ackermann($m$, $n - 1$))
```

Once you've designed your function, test it by calling it with small values for `m` and `n`.

CHAPTER 12 GUI Programming

TOPICS

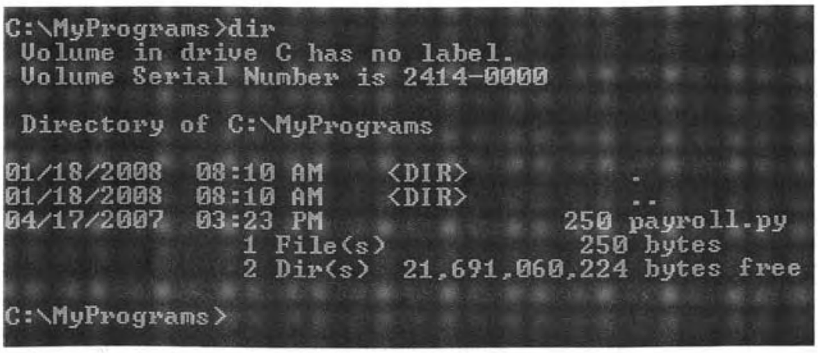
12.1 Graphical User Interfaces
12.2 Using the **`Tkinter`** Module
12.3 Display Text with **`Label`** Widgets
12.4 Organizing Widgets with Frames
12.5 Button Widgets and Info Dialog Boxes
12.6 Getting Input with the **`Entry`** Widget
12.7 Using Labels as Output Fields
12.8 Radio Buttons and Check Buttons

12.1 Graphical User Interfaces

CONCEPT: **A** graphical user interface allows the user to interact with the operating system and other programs using graphical elements such as icons, buttons, and dialog boxes.

A computer's *user interface* is the part of the computer that the user interacts with. One part of the user interface consists of hardware devices, such as the keyboard and the video display. Another part of the user interface lies in the way that the computer's operating system accepts commands from the user. For many years, the only way that the user could interact with an operating system was through a *command line interface,* such as the one shown in Figure 12-1. A command line interface typically displays a prompt, and the user types a command, which is then executed.

Figure 12-1 A command line interface



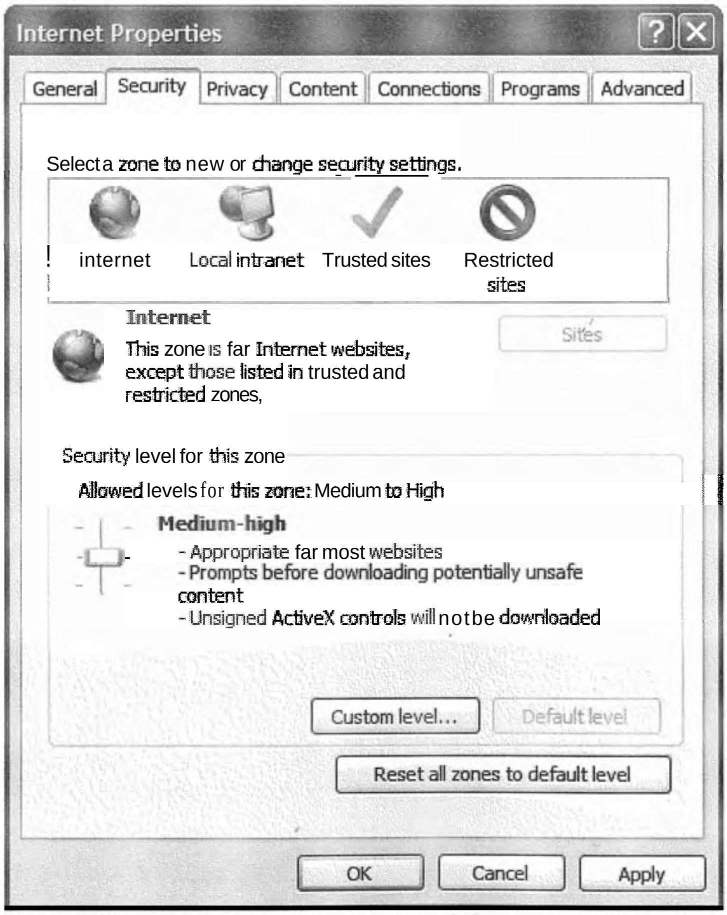
```
C:\MyPrograms>dir
 Volume in drive C has no label.
 Volume Serial Number is 2414-0000

 Directory of C:\MyPrograms

01/18/2008  08:10 AM    <DIR>          .
01/18/2008  08:10 AM    <DIR>          ..
04/17/2007  03:23 PM               250 payroll.py
               1 File(s)            250 bytes
               2 Dir(s)  21,691,060,224 bytes free

C:\MyPrograms>
```

Many computer users, especially beginners, find command line interfaces difficult to use. This is because there are many commands to be learned, and each command has its own syntax, much like a programming statement. If a command isn't entered correctly, it will not work.

In the 1980s, a new type of interface known as a graphical user interface came into use in commercial operating systems. A *graphical user interface (GUI)* (pronounced "gooey"), allows the user to interact with the operating system and other programs through graphical elements on the screen. GUIs also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them.

Much of the interaction with a GUI is done through *dialog boxes*, which are small windows that display information and allow the user to perform actions. Figure 12-2 shows an example of a dialog box from the Windows operating system that allows the user to change the system's Internet settings. Instead of typing commands according to a specified syntax, the user interacts with graphical elements such as icons, buttons, and slider bars.

Figure 12-2 A dialog **box**


Internet Properties
General
Security
Privacy
Content
Connections
Programs
Advanced
internet
Local intranet
Trusted sites
Restricted sites
Sites
Custom level...
Default level
Reset all zones to default level
OK
Cancel
Apply

Select a zone to new or change security settings.

Internet

This zone is far Internet websites, except those listed in trusted and restricted zones,

Security level for this zone

Allowed levels for this zone: Medium to High

Medium-high

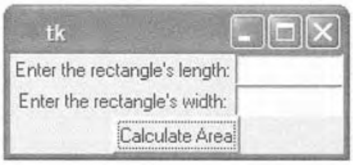
- Appropriate far most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

GUI Programs Are Event-Driven

In a text-based environment, such as a command line interface, programs determine the order in which things happen. For example, consider a program that calculates the area of a rectangle. First, the program prompts the user to enter the rectangle's width. The user enters the width and then the program prompts the user to enter the rectangle's length. The user enters the length and then the program calculates the area. The user has no choice but to enter the data in the order that it is requested.

In a GUI environment, however, the user determines the order in which things happen. For example, Figure 12-3 shows a GUI program (written in Python) that calculates the area of a rectangle. The user can enter the length and the width in any order he or she wishes. If a mistake is made, the user can erase the data that was entered and retype it. When the user is ready to calculate the area, he or she clicks the *Calculate Area* button and the program performs the calculation. Because GUI programs must respond to the actions of the user, it is said that they are *event-driven*. The user causes events to take place, such as the clicking of a button, and the program must respond to the events.

Figure 12-3 A GUI program



Enter the rectangle's length:
Enter the rectangle's width:

Checkpoint

12.1 What is a user interface?

12.2 How does a command line interface work?

12.3 When the user runs a program in a text-based environment, such as the command line, what determines the order in which things happen?

12.4 What is an event-driven program?

12.2 Using the Tkinter Module

CONCEPT: In Python you can use the **`Tkinter`** module to create simple GUI programs.

Python does not have GUI programming features built into the language itself. However, it comes with a module named `Tkinter` that allows you to create simple GUI programs. The name "Tkinter" is short for "Tk interface." It is named this because it provides a way for Python programmers to use a GUI library named Tk. Many other programming languages use the Tk library as well.

> **NOTE:** There are numerous GUI libraries available for Python. Because the `Tkinter` module comes with Python, we will use it only in this chapter.

A GUI program presents a window with various graphical ***widgets*** that the user can interact with or view. The `Tkinter` module provides 15 widgets, which are described in Table 12-1. We won't cover all of the `Tkinter` widgets in this chapter, but we will demonstrate how to create simple GUI programs that gather input and display data.

Table 12-1 Tkinter Widgets

Widget	Description
`Button`	A button that can cause an action to occur when it is clicked.
`Canvas`	A rectangular area that can be used to display graphics.
`Checkbutton`	A button that may be in either the "on" or "off" position.
`Entry`	An area in which the user may type a single line of input from the keyboard.
`Frame`	A container that can hold other widgets.
`Label`	An area that displays one line of text or an image.
`Listbox`	A list from which the user may select an item
Menu	A list of menu choices that are displayed when the user clicks a `Menubutton` widget.
`Menubutton`	A menu that is displayed on the screen and may be clicked by the user
`Message`	Displays multiple lines of text.
`Radiobutton`	A widget that can be either selected or deselected. Radiobuttons usually appear in groups and allow the user to select one of several options.
`Scale`	A widget that allows the user to select a value by moving a slider along a track.
`Scrollbar`	Can be used with some other types of widgets to provide scrolling ability.
`Text`	A widget that allows the user to enter multiple lines of text input.
`Toplevel`	A container, like a `Frame`, but displayed in its own window.

The simplest GUI program that we can demonstrate is one that displays an empty window. Program 12-1 shows how we can do this using the `Tkinter` module. When the program runs, the window shown in Figure 12-4 is displayed. To exit the program, simply click the standard Windows close button (☒) in the upper right corner of the window.

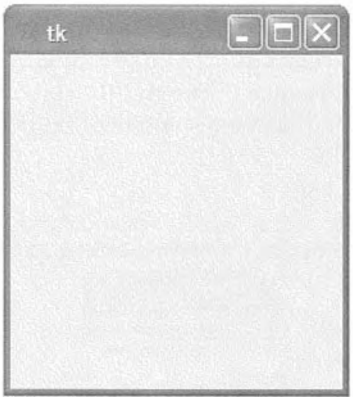
NOTE: Programs that use `Tkinter` do not always run reliably under IDLE. This is because IDLE itself uses `Tkinter`. You can always use IDLE's editor to write GUI programs, but for the best results, run them from your operating system's command prompt.

Program 12-1 (empty_window1.py)

```
# This program displays an empty window.

```

```
 3  import Tkinter

 5  def main():
 6      # Create the main window widget.
 7      main-window = Tkinter.Tk()
 8
 9      # Enter the Tkinter main loop.
10      Tkinter.mainloop()
11
12  # Call the main function.
13  main()
```

Figure 12-4 Window displayed by Program 12-1



Line 3 imports the `Tkinter` module. Inside the `main` function, line 7 creates an instance of the `Tkinter` module's `Tk` class, and assigns it to the `main_window` variable. This object is the root widget, which is the main window in the program. Line 10 calls the `Tkinter` module's `mainloop` function. This function runs like an infinite loop until you close the main window.

Most programmers prefer to take an object-oriented approach when writing a GUI program. Rather than writing a function to create the on-screen elements of a program, it is a common practice to write a class with an `__init__` method that builds the GUI. When an instance of the class is created, the GUI appears on the screen. To demonstrate, Program 12-2 shows an object-oriented version of our program that displays an empty window. When this program runs it displays the window shown in Figure 12-4.

Program 12-2 (empty_window2.py)

```
# This program displays an empty window.

import Tkinter
```

(program continues)

Program 12-2 *(continued)*

```
 5  class MyGUI:
 6      def __init__(self):
 7          # Create the main window widget.
 8          self.main_window = Tkinter.Tk()
 3
10          # Enter the Tkinter main loop.
11          Tkinter.mainloop()
12
13  # Create an instance of the MyGUI class.
14  my_gui = MyGUI()
```

Lines 5 through 11 are the class definition for the `MyGUI` class. The class's `__init__` method begins in line 6. Line 8 creates the root widget and assigns it to the class attribute `main_window`. Line 11 executes the `Tkinter` module's `mainloop` function. The statement in line 14 creates an instance of the `MyGUI` class. This causes the class's `__init__` method to execute, displaying the empty window on the screen.

Checkpoint

12.5 Briefly describe each of the following `Tkinter` widgets:

a) `Label`
b) `Entry`
c) `Button`
d) `Frame`

12.6 How do you create a root widget?

12.7 What does the `Tkinter` module's `mainloop` function do?

12.3 Display Text with `Label` Widgets

CONCEPT: You use the `Label` widget to display text in a window.

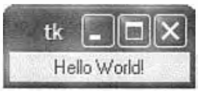
You can use a `Label` widget to display a single line of text in a window. To make a `Label` widget you create an instance of the `Tkinter` module's `Label` class. Program 12-3 creates a window containing a `Label` widget that displays the text "Hello World!" The window is shown in Figure 12-5.

Program 12-3 (hello_world.py)

```
# This program displays a label with text.

import Tkinter

```

```
 5  class MyGUI:
 6      def __init__(self):
 7          # Create the main window widget.
 8          self.main_window = Tkinter.Tk()
 9
10          # Create a Label widget containing the
11          # text 'Hello World!'
12          self.label = Tkinter.Label(self.main_window, \
13                                     text='Hello World!')
14
15          # Call the Label widget's pack method.
16          self.label.pack()
17
18          # Enter the Tkinter main loop.
19          Tkinter.mainloop()
20
21  # Create an instance of the MyGUI class.
22  my_gui = MyGUI()
```

Figure 12-5 Window displayed by Program 12-3



Hello World!

The `MyGUI` class in this program is very similar to the one you saw previously in Program 12-2. Its `__init__` method builds the GUI when an instance of the class is created. Line 8 creates a root widget and assigns it to `self.main_window`. The following statement appears in lines 12 and 13:

```
self.label = Tkinter.Label(self.main_window, \
                           text='Hello World!')
```

This statement creates a `Label` widget and assigns it to `self.label`. The first argument inside the parentheses is `self.main_window`, which is a reference to the root widget. This simply specifies that we want the `Label` widget to belong to the root widget. The second argument is `text='Hello World!'`. This specifies the text that we want displayed in the label.

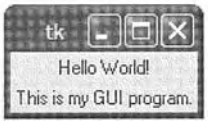
The statement in line 16 calls the `Label` widget's `pack` method. The `pack` method determines where a widget should be positioned, and makes the widget visible when the main window is displayed. (You call the `pack` method for each widget in a window.) Line 19 calls the `Tkinter` module's `mainloop` method which displays the program's main window, shown in Figure 12-5.

Let's look at another example. Program 12-4 displays a window with two `Label` widgets, shown in Figure 12-6.

Program 12-4 (hello_world2.py)

```
# This program displays two labels with text.

import Tkinter

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = Tkinter.Tk()

        # Create two Label widget.
        self.label1 = Tkinter.Label(self.main_window, \
                                    text='Hello World!')
        self.label2 = Tkinter.Label(self.main_window, \
                        text='This is my GUI program.')

        # Call both Label widgets' pack method.
        self.label1.pack()
        self.label2.pack()

        # Enter the Tkinter main loop.
        Tkinter.mainloop()

# Create an instance of the MyGUI class.
my_gui = MyGUI()
```

Figure 12-6 Window displayed by Program 12-4



Hello World!

This is my GUI program.

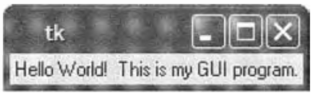
Notice that the two `Label` widgets are displayed with one stacked on top of the other. We can change this layout by specifying an argument to `pack` method, as shown in Program 12-5. When the program runs it displays the window shown in Figure 12-7.

Program 12-5 (hello_world3.py)

```
# This program uses the side='left' argument with
# the pack method to change the layout of the widgets.

import Tkinter

class MyGUI:
    def __init__(self):
```

```
 8          # Create the main window widget.
 9          self.main_window = Tkinter.Tk()
10
11          # Create two Label widgets.
12          self.label1 = Tkinter.Label(self.main_window, \
13                                      text='Hello World!')
14          self.label2 = Tkinter.Label(self.main_window, \
15                          text='This is my GUI program.')
16
17          # Call both Label widgets' pack method.
18          self.label1.pack(side='left')
19          self.label2.pack(side='left')
20
21          # Enter the Tkinter main loop.
22          Tkinter.mainloop()
23
24  # Create an instance of the MyGUI class.
25  my-gui = MyGUI()
```

Figure 12-7 Window displayed by Program 12-5



Hello World! This is my GUI program.

In lines 18 and 19 we call each `Label` widget's `pack` method passing the argument `side='left'`. This specifies that the widget should be positioned as far left as possible inside the parent widget. Because the `label1` widget was added to the `main_window` first, it will appear at the leftmost edge. The `label2` widget was added next, so it appears next to the `label1` widget. As a result, the labels appear side by side. The valid `side` arguments that you can pass to the `pack` method are `side='top'`, `side='bottom'`, `side='left'`, and `side='right'`.

Checkpoint

12.8 What does a widget's `pack` method do?

12.9 If you create two `Label` widgets and call their `pack` methods with no arguments, how will the `Label` widgets be arranged inside their parent widget?

12.10 What argument would you pass to a widget's `pack` method to specify that it should be positioned as far left as possible inside the parent widget?

12.4 Organizing Widgets with Frames

CONCEPT: **A `Frame` is a container that can hold other widgets. You can use `Frames` to organize the widgets in a window.**

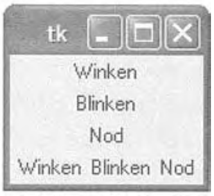
A Frame is a container. It is a widget that can hold other widgets. Frames are useful for organizing and arranging groups of widgets in a window. For example, you can place a set of widgets in one Frame and arrange them in a particular way, then place a set of widgets in another Frame and arrange them in a different way. Program 12-6 demonstrates this. When the program runs it displays the window shown in Figure 12-8.

Program 12-6 (frame_demo.py)

```
# This program creates labels in two different frames.

import Tkinter

class MyGUI:
    def __init--(self):
        # Create the main window widget.
        self·main-window = Tkinter.Tk()

        # Create two frames, one for the top of the
        # window, and one for the bottom.
        self·top-frame = Tkinter.Frame(self.main_window)
        self.bottom_frame = Tkinter.Frame(self.main_window)

        # Create three Label widgets for the
        # top frame.
        self.label1 = Tkinter.Label(self.top_frame, \
                                    text='Winken')
        self.label2 = Tkinter.Label(self.top_frame, \
                                    text='Blinken')
        self.label3 = Tkinter.Label(self.top_frame, \
                                    text='Nod')

        # Pack the labels that are in the top frame.
        # Use the side='top' argument to stack them
        # one on top of the other.
        self.label1.pack(side='top')
        self.label2.pack(side='top')
        self.label3.pack(side='top')

        # Create three Label widgets .for the
        # bottom frame.
        self.label4 = Tkinter.Label(self.top_frame, \
                                    text='Winken')
        self.label5 = Tkinter.Label(self.top_frame, \
                                    text='Blinken')
        self.label6 = Tkinter.Label(self.top_frame, \
                                    text='Nod')

```

```
40          # Pack the labels that are in the bottom frame.
41          # Use the side='left' argument to arrange them
42          # horizontally from the left of the frame.
43          self.label4.pack(side='left')
44          self.label5.pack(side='left')
45          self.label6.pack(side='left')
46
47          # Yes, we have to pack the frames too!
48          self.top_frame.pack()
49          self.bottom_frame.pack()
50
51          # Enter the Tkinter main loop.
52          Tkinter.mainloop()
53
54  # Create an instance of the MyGUI class.
55  my_gui = MyGUI()
```

Figure 12-8 Window displayed by Program 12-6



Take a closer look at lines 12 and 13:

```
self.top_frame = Tkinter.Frame(self.main_window)
self.bottom_frame = Tkinter.Frame(self.main_window)
```

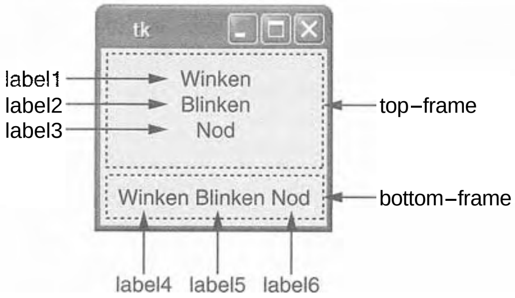
These, lines create two `Frame` objects. The `self.main_window` argument that appears inside the parentheses cause the `Frames` to be added to the `main_window` widget.

Lines 17 through 22 create three `Label` widgets. Notice that these widgets are added to the `self.top_frame` widget. Then, lines 27 through 29 call each of the `Label` widgets' `pack` method, passing `side='top'` as an argument. As shown in Figure 12-6, this causes the three widgets to be stacked one on top of the other inside the `Frame.`

Lines 23 through 28 create three more `Label` widgets. These `Label` widgets are added to the `self.bottom_frame` widget. Then, lines 43 through 45 call each of the `Label` widgets' `pack` method, passing `side='left'` as an argument. As shown in Figure 12-9, this causes the three widgets to appear horizontally inside the `Frame.`

Lines 48 and 49 call the `Frame` widgets' `pack` method, which makes the `Frame` widgets visible. Line 52 executes the `Tkinter` module's `mainloop` function.

Figure 12-9 Arrangement of widgets



12.5 Button Widgets and Info Dialog Boxes

CONCEPT: You use the **`Button`** widget to create a standard button in a window. When the user clicks a button, a specified function or method is called.

An info dialog box is a simple window that displays a message to the user and has an OK button that dismisses the dialog box. You can use the **`tkMessageBox`** module's **`showinfo`** function to display an info dialog box.

A `Button` is a widget that the user can click to cause an action to take place. When you create a `Button` widget you can specify the text that is to appear on the face of the button, and the name of a callback function. A *callback function* is a function or method that executes when the user clicks the button.

NOTE: A callback function is also known as an *event handler* because it handles the event that occurs when the user clicks the button.

To demonstrate, we will look at Program 12-7. This program displays the window shown in Figure 12-10. When the user clicks the button, the program displays a separate *info dialog box*, shown in Figure 12-11. We use a function named `showinfo`, which is in the `tkMessageBox` module, to display the info dialog box. (To use the `showinfo` function you will need to import the `tkMessageBox` module.) This is the general format of the `showinfo` function call:

```
tkMessageBox.showinfo(title, message)
```

In the general format, *`title`* is a string that is displayed in the dialog box's title bar, and *`message`* is an informational string that is displayed in the main part of the dialog box.

Program 12-7 (button_demo.py)

```
# This program demonstrates a Button widget.
# When the user clicks the Button, an
# info dialog box is displayed.
```

```

import Tkinter
import tkMessageBox

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = Tkinter.Tk()

        # Create a Button widget. The text 'Click Me!'
        # should appear on the face of the Button. The
        # do-something method should be executed when
        # the user clicks the Button.
        self.my_button = Tkinter.Button(self.main_window, \
                     text='Click Me!', command=self.do_something)

        # Pack the Button.
        self.my_button.pack()

        # Enter the Tkinter main loop.
        Tkinter.mainloop()

    # The do-something method is a callback function
    # for the Button widget.

    def do_something(self):
        # Display an info dialog box.
        tkMessageBox.showinfo('Response', \
                    'Thanks for clicking the button.')

# Create an instance of the MyGUI class.
my-gui = MyGUI()
```

Figure 12-10 The main window displayed by Program 12-7

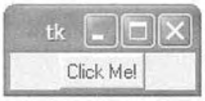
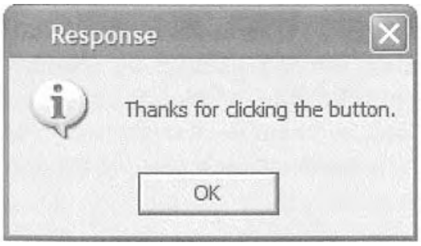


Figure 12-11 The info dialog box displayed by Program 12-7



Thanks for clicking the button.

Line 5 imports the `Tkinter` module and line 6 imports the `tkMessageBox` module. Line 11 creates the root widget and assigns it to the `main_window` variable.

The statement in lines 17 through 18 creates the `Button` widget. The first argument inside the parentheses is `self.main_window`, which is the parent widget. The `text='Click Me!'` argument specifies that the string 'Click Me!' should appear on the face of the button. The command=`'self.do_something'` argument specifies the class's `do_something` method as the callback function. When the user clicks the button, the `do_something` method will execute.

The `do_something` method appears in lines 29 through 32. The method simply calls the `tkMessageBox.showinfo` function to display the info box shown in Figure 12-11. To dismiss the dialog box the user can click the OK button.

Creating a Quit Button

GUI programs usually have a *Quit button* (or an Exit button) that closes the program when the user clicks it. To create a Quit button in a Python program you simply create a Button widget that calls the root widget's `quit` method as a callback function. Program 12-8 demonstrates how to do this. It is a modified version of Program 12-7, with a second `Button` widget added as shown in Figure 12-12.

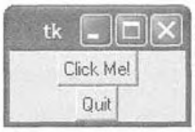
> **NOTE:** All widgets inherit the `quit` method, so you can use any widget in your program to call it. It is somewhat customary, however, to use the root widget to call the `quit` method.

Program 12-8 (quit_button.py)

```
# This program has a Quit button that calls
# the Tk class's quit method when clicked.

import Tkinter
import tkMessageBox

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = Tkinter.Tk()

        # Create a Button widget. The text 'Click Me!'
        # should appear on the face of the Button. The
        # do_something method should be executed when
        # the user clicks the Button.
        self.my_button = Tkinter.Button(self.main_window, \
                               text='Click Me!', command=self.do_something)

```

```
19          # Create a Quit button. When this button is clicked
20          # the root widget's quit method is called.
21          # (The main-window variable references the root widget,
22          # so the callback function is self.main_window.quit.)
23          self.quit_button = Tkinter.Button(self.main_window, \
24                        text='Quit', command=self.main_window.quit)
25
26          # Pack the Buttons.
27          self.my_button.pack()
28          self.quit_button.pack()
29
30          # Enter the Tkinter main loop.
31          Tkinter.mainloop()
32
33      # The do-something method is a callback function
34      # for the Button widget.
35
36      def do_something(self):
37          # Display an info dialog box.
38          tkMessageBox.showinfo('Response', \
39                      'Thanks for clicking the button.')
40
41  # Create an instance of the MyGUI class.
42  my_gui = MyGUI()
```

Figure 12-12 The info dialog box displayed by Program 12-7



The statement in lines 23 through 24 creates the Quit button. Notice that the `self.main_window.quit` method is used as the callback function. When the user clicks the button, this method is called and the program ends.

12.6 Getting Input with the `Entry` Widget

CONCEPT: **An `Entry` widget is a rectangular area that the user can type input into. You use the `Entry` widget's get method to retrieve the data that has been typed into the widget.**

An **`Entry`** widget is a rectangular area that the user can type text into. Entry widgets are used to gather input in a GUI program. Typically, a program will have one or more `Entry` widgets in a window, along with a button that the user clicks to submit the data that he or she has typed into the `Entry` widgets. The button's callback function retrieves data from the window's `Entry` widgets and processes it.

You use an `Entry` widget's `get` method to retrieve the data that the user has typed into the widget. The `get` method returns a string, so it will have to be converted to the appropriate data type if the `Entry` widget is used for numeric input.

To demonstrate we will look at a program that allows the user to enter a distance in kilometers into an `Entry` widget, and then click a button to see that distance converted to miles. The formula for converting kilometers to miles is:

$$Miles = Kilometers \times 0.6214$$

Figure 12-13 shows the window that the program displays. To arrange the widgets in the positions shown in the figure, we will organize them in two frames, as shown in Figure 12-14. The label that displays the prompt and the `Entry` widget will be stored in the `top_frame`, and their `pack` methods will be called with the `side='left'` argument. This will cause them to appear horizontally in the frame. The Convert button and the Quit button will be stored in the `bottom_frame`, and their `pack` methods will also be called with the `side='left'` argument.

Program 12-9 shows the code for the program. Figure 12-15 shows what happens when the user enters 1000 into the `Entry` widget and then clicks the Convert button.

Figure 12-13 The `kilo_converter` program's window

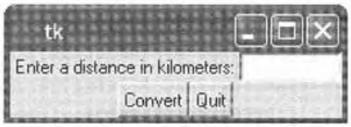
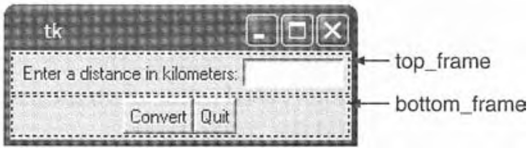


Figure 12-14 The window organized with frames



Program 12-9 (kilo_converter.py)

```
# This program converts distances in kilometers
# to miles. The result is displayed in an info
# dialog box.

import Tkinter
import tkMessageBox
```

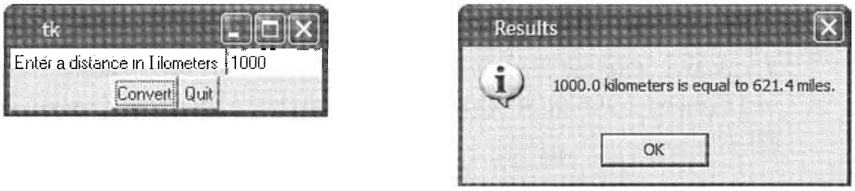
```

class KiloConverterGUI:
    def —init—(self):

        # Create the main window.
        self·main-window = Tkinter.Tk()

        # Create two frames to group widgets.
        self.top_frame = Tkinter.Frame(self.main_window)
        self.bottom_frame = Tkinter.Frame(self.main_window)

        # Create the widgets for the top frame.
        self·prompt-label = Tkinter.Label(self.top_frame, \
                    text='Enter a distance in kilometers:')
        self.kilo_entry = Tkinter.Entry(self.top_frame, \
                                        width=10)

        # Pack the top frame's widgets.
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        # Create the button widgets for the bottom frame.
        self.calc_button = Tkinter.Button(self.bottom_frame, \
                                      text='Convert', \
                                      command=self.convert)
        self.quit_button = Tkinter.Button(self.bottom_frame, \
                                text='Quit', \
                                command=self.main_window.quit)
        # Pack the buttons.
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.top_frame.pack()
        self.bottom_frame.pack()

        # Enter the Tkinter main loop.
        Tkinter.mainloop()

    # The convert method is a callback function for
    # the Calculate button.

    def convert(self):
        # Get the value entered by the user into the
        # kilo-entry widget.
        kilo = float(self.kilo_entry.get())

```

(program continues)

Program 12-9 *(continued)*

```
54          # Convert kilometers to miles.
55          miles = kilo * 0.6214
56
57          # Display the results in an info dialog box.
58          tkMessageBox.showinfo('Results', \
59              str(kilo) + ' kilometers is equal to ' + \
60              str(miles) + ' miles.')
61
62  # Create an instance of the KiloConverterGUI class.
63  kilo_conv = KiloConverterGUI()
```

Figure 12-15 The info dialog box



Enter a distance in kilometers 1000

1000.0 kilometers is equal to 621.4 miles.

① The user enters 1000 into the Entry widget and clicks the Convert button.

② This info dialog box is displayed.

The `convert` method, shown in lines 49 through 60 is the Convert button's callback function. The statement in line 52 calls the `kilo_entry` widget's `get` method to retrieve the data that has been typed into the widget. The value is converted to a `float` and then assigned to the `kilo` variable. The calculation in line 55 performs the conversion and assigns the results to the `miles` variable. Then, the statement in lines 58 through 60 displays the info dialog box with a message that gives the converted value.

12.7 Using Labels as Output Fields

CONCEPT: When a **`StringVar`** object is associated with a **`Label`** widget, the **`Label`** widget displays any data that is stored in the **`StringVar`** object.

Previously you saw how to use an info dialog box to display output. If you don't want to display a separate dialog box for your program's output, you can use `Label` widgets in the program's main window to dynamically display output. You simply create empty `Label` widgets in your main window, and then write code that displays the desired data in those labels when a button is clicked.

The `Tkinter` module provides a class named `StringVar` that can be used along with a `Label` widget to display data. First you create a `StringVar` object. Then, you create a

`Label` widget and associate it with the `StringVar` object. From that point on, any value that is then stored in the `StringVar` object will automatically be displayed in the `Label` widget.

Program 12-10 demonstrates how to do this. It is a modified version of the `kilo_converter` program that you saw in Program 12-9. Instead of popping up an info dialog box, this version of the program displays the number of miles in a label in the main window.

Program 12-10 (kilo_converter2.py)

```
# This program converts distances in kilometers
# to miles. The result is displayed in a label
# on the main window.

import Tkinter

class KiloConverterGUI:
    def __init__(self):

        # Create the main window.
        self.main_window = Tkinter.Tk()

        # Create three frames to group widgets.
        self.top_frame = Tkinter.Frame()
        self.mid_frame = Tkinter.Frame()
        self.bottom_frame = Tkinter.Frame()

        # Create the widgets for the top frame.
        self.prompt_label = Tkinter.Label(self.top_frame, \
                    text='Enter a distance in kilometers:')
        self.kilo_entry = Tkinter.Entry(self.top_frame, \
                                        width=10)

        # Pack the top frame's widgets.
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        # Create the widgets for the middle frame.
        self.descr_label = Tkinter.Label(self.mid_frame, \
                                 text='Converted to miles:')

        # We need a StringVar object to associate with
        # an output label. Use the object's set method
        # to store a string of blank characters.
        self.value = Tkinter.StringVar()

```

(program continues)

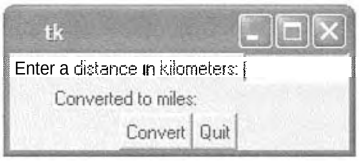
Program 12-10 *(continued)*

```
        # Create a label and associate it with the
        # StringVar object. Any value stored in the
        # StringVar object will automatically be displayed
        # in the label.
        self.miles_label = Tkinter.Label(self.mid_frame, \
                                   textvariable=self.value)

        # Pack the middle frame's widgets.
        self.descr_label.pack(side='left')
        self.miles_label.pack(side='left')

        # Create the button widgets for the bottom frame.
        self.calc_button = Tkinter.Button(self.bottom_frame, \
                                     text='Convert', \
                                     command=self.convert)
        self.quit_button = Tkinter.Button(self.bottom_frame, \
                                text='Quit', \
                                command=self.main_window.quit)

        # Pack the buttons.
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.top_frame.pack()
        self.mid_frame.pack()
        self.bottom_frame.pack()

        # Enter the Tkinter main loop.
        Tkinter.mainloop()

    # The convert method is a callback function for
    # the Calculate button.

    def convert(self):
        # Get the value entered by the user into the
        # kilo-entry widget.
        kilo = float(self.kilo_entry.get())

        # Convert kilometers to miles.
        miles = kilo * 0.6214

        # Convert miles to a string and store it
        # in the StringVar object. This will automatically
        # update the miles-label widget.
```

```
82          self.value.set(miles)
83
84  # Create an instance of the KiloConverterGUI class.
85  kilo_conv = KiloConverterGUI()
```

When this program runs it displays the window shown in Figure 12-16. Figure 12-17 shows what happens when the user enters 1000 for the kilometers and clicks the Convert button. The number of miles is displayed in a label in the main window.

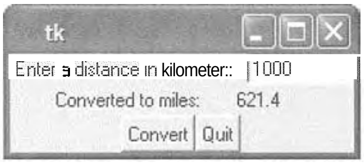
Figure 12-16 The window initially displayed



Enter a distance in kilometers:

Converted to miles:

Figure 12-17 The window showing 1000 kilometers converted to miles



Enter a distance in kilometers: 1000

Converted to miles: 621.4

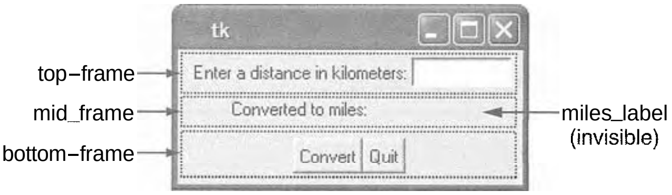
Let's look at the code. Lines 14 through 16 create three frames: `top_frame`, `mid_frame`, and `bottom_frame`. Lines 19 through 26 create the widgets for the top frame and calls their `pack` method.

Lines 29 through 30 create the Label widget with the text `'Converted to miles:'` that you see on the main window in Figure 12-16. Then, line 35 creates a `StringVar` object and assigns it to the `value` variable. Line 41 creates a `Label` widget named `miles_label` that we will use to display the number of miles. Notice that in line 42 we use the argument `textvariable=self.value`. This creates an association between the `Label` widget and the `StringVar` object that is referenced by the `value` variable. Any value that we store in the `StringVar` object will be displayed in the label.

Lines 45 and 46 pack the two `Label` widgets that are in the `mid_frame`, Lines 49 through 58 create the `Button` widgets and `pack` them. Lines 61 through 63 pack the Frame objects. Figure 12-18 shows how the various widgets in this window are organized in the three frames.

The `convert` method, shown in lines 71 through 82 is the Convert button's callback function. The statement in line 74 calls the `kilo_entry` widget's `get` method to retrieve the data that has been typed into the widget. The value is converted to a `float` and then assigned to the `kilo` variable. The calculation in line 77 performs the conversion and assigns the results to the `miles` variable. Then the statement in line 82 calls the `StringVar` object's `set` method, passing `miles` as an argument. This stores the value referenced by `miles` in the `StringVar` object, and also causes it to be displayed in the `miles_label` widget.

Figure 12-18 Layout of the `kilo_converter2` program's `main` window


tk
top-frame
mid_frame
bottom-frame
Enter a distance in kilometers:
Converted to miles:
miles_label
(invisible)
Convert
Quit

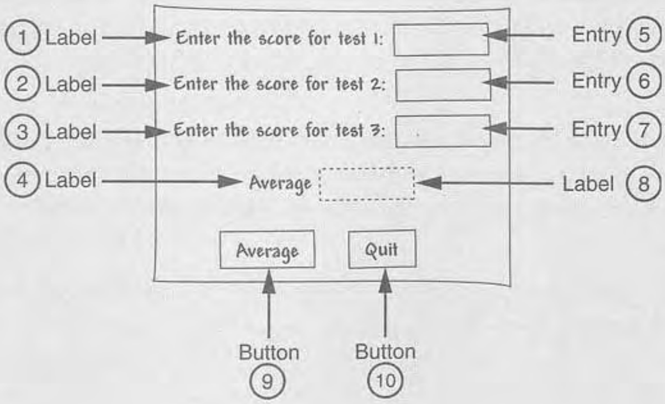
In the Spotlight:

Creating a GUI Program

Kathryn teaches a science class. In Chapter 4, we stepped through the development of a program that her students can use to calculate the average of three test scores. The program prompts the student to enter each score, and then it displays the average. She has asked you to design a GUI program that performs a similar operation. She would like the program to have three `Entry` widgets that the test scores can be entered into, and a button that causes the average to be displayed when clicked.

Before we begin writing code, it will be helpful if we draw a sketch of the program's window, as shown in Figure 12-19. The sketch also shows the type of each widget. (The numbers that appear in the sketch will help us when we make a list of all the widgets.)

Figure 12-19 A sketch of the window

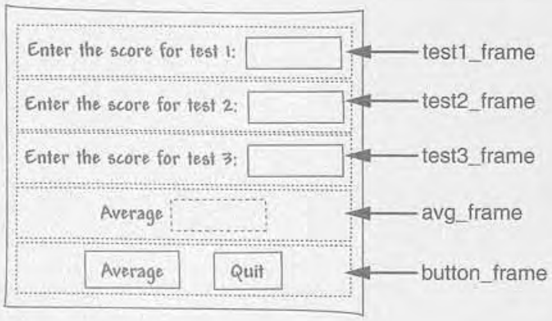

① Label
② Label
③ Label
④ Label
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Entry ⑤
Entry ⑥
Entry ⑦
Label ⑧
Average
Quit
Button ⑨
Button ⑩

By examining the sketch we can make a list of the widgets that we need. As we make the list, we will include a brief description of each widget and a name that we will assign to each widget when we construct it.

Widget Number in Figure 12-19	Widget Type	Description	Name
1	Label	Instructs the user to enter the score for test 1.	`test1_label`
2	Label	Instructs the user to enter the score for test 2.	`test2_label`
3	Label	Instructs the user to enter the score for test 3.	`test3_label`
4	Label	Identifies the average, which will be displayed next to this label.	`result_label`
5	Entry	This is where the user will enter the score for test 1.	`test1_entry`
6	Entry	This is where the user will enter the score for test 2.	`test2_entry`
7	Entry	This is where the user will enter the score for test 3.	`test3_entry`
8	Label	The program will display the average test score in this label.	`avg_label`
9	Button	When this button is clicked, the program will calculate the average test score and display it in the `averageLabel` component.	`calc_button`
10	Button	When this button is clicked the program will end.	`quit_button`

We can see from the sketch that we have five rows of widgets in the window. To organize them we will also create five `Frame` objects. Figure 12-20 shows how we will position the widgets inside the five `Frame` objects.

Figure 12-20 Using frames to organize the widgets

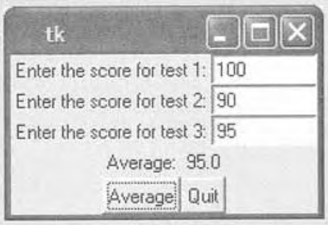

Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Average
Quit
test1_frame
test2_frame
test3_frame
avg_frame
button_frame

Program 12-11 shows the code for the program, and Figure 12-21 shows the program's window with data entered by the user.

Program 12-11 (test_averages.py)

```
# This program uses a GUI to get three test
# scores and display their average.

import Tkinter

class TestAvg:
    def __init__(self):
        # Create the main window.
        self.main_window = Tkinter.Tk()

        # Create the five frames.
        self.test1_frame = Tkinter.Frame(self.main_window)
        self.test2_frame = Tkinter.Frame(self.main_window)
        self.test3_frame = Tkinter.Frame(self.main_window)
        self.avg_frame = Tkinter.Frame(self.main_window)
        self.button_frame = Tkinter.Frame(self.main_window)

        # Create and pack the widgets for test 1.
        self.test1_label = Tkinter.Label(self.test1_frame, \
                        text='Enter the score for test 1:')
        self.test1_entry = Tkinter.Entry(self.test1_frame, \
                                         width=10)
        self.test1_label.pack(side='left')
        self.test1_entry.pack(side='left')

        # Create and pack the widgets for test 2.
        self.test2_label = Tkinter.Label(self.test2_frame, \
                        text='Enter the score for test 2:')
        self.test2_entry = Tkinter.Entry(self.test2_frame, \
                                         width=10)
        self.test2_label.pack(side='left')
        self.test2_entry.pack(side='left')

        # Create and pack the widgets for test 3.
        self.test3_label = Tkinter.Label(self.test3_frame, \
                        text='Enter the score for test 3:')
        self.test3_entry = Tkinter.Entry(self.test3_frame, \
                                         width=10)
        self.test3_label.pack(side='left')
        self.test3_entry.pack(side='left')

        # Create and pack the widgets for the average.
        self.result_label = Tkinter.Label(self.avg_frame, \
                        text='Average:')
        self.avg = Tkinter.StringVar() # To update avg_label
        self.avg_label = Tkinter.Label(self.avg_frame, \
                                    textvariable=self.avg)
        self.result_label.pack(side='left')
        self.avg_label.pack(side='left')

        # Create and pack the button widgets.
        self.calc_button = Tkinter.Button(self.button_frame, \
                                     text='Average', \
                                     command=self.calc_avg)
        self.quit_button = Tkinter.Button(self.button_frame, \
                                recommand=self.main_window.quit)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.test1_frame.pack()
        self.test2_frame.pack()
        self.test3_frame.pack()
        self.avg_frame.pack()
        self.button_frame.pack()

        # Start the main loop.
        Tkinter.mainloop()

    # The calc_avg method is the callback function for
    # the calc_button widget.

    def calc_avg(self):
        # Get the three test scores and store them
        # in variables.
        self.test1 = float(self.test1_entry.get())
        self.test2 = float(self.test2_entry.get())
        self.test3 = float(self.test3_entry.get())

        # Calculate the average.
        self.average = (self.test1 + self.test2 + \
                       self.test3) / 3.0

        # Update the avg_label widget by storing
        # the value of self.average in the StringVar
        # object referenced by avg.
        self.avg.set(self.average)

# Create an instance of the TestAvg class.
test_avg = TestAvg()
```

Figure 12-21 The `test_averages` program window



Enter the score for test 1: 100
Enter the score for test 2: 90
Enter the score for test 3: 95
Average: 95.0

Checkpoint

12.11 How do you retrieve data from an `Entry` widget?

12.12 When you retrieve a value from an `Entry` widget, of what data type is it?

12.13 What module is the `StringVar` class in?

12.14 What can you accomplish by associating a `StringVar` object with a `Label` widget?

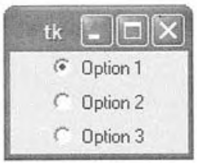
12.8 Radio Buttons and Check Buttons

CONCEPT: Radio buttons normally appear in groups of two or more and allow the user to select one of several possible options. Check buttons, which may appear alone or in groups, allow the user to make yes/no or on/off selections.

Radio Buttons

Radio buttons are useful when you want the user to select one choice from several possible options. Figure 12-22 shows a group of radio buttons. A radio button may be selected or deselected. Each radio button has a small circle that appears filled in when the radio button is selected and appears empty when the radio button is deselected.

Figure 12-22 A group of radio buttons



You use the `Tkinter` module's `Radiobutton` class to create `Radiobutton` widgets. Only one of the `Radiobutton` widgets in a container, such as a frame, may be selected at any time. Clicking a `Radiobutton` selects it and automatically deselects any other `Radiobutton` in the same container. Because only one `Radiobutton` in a container can be selected at any given time, they are said to be *mutually exclusive.*

NOTE: The name "radio button" refers to the old car radios that had push buttons for selecting stations. Only one of the buttons could be pushed in at a time. When you pushed a button in, it automatically popped out any other button that was pushed in.

The `Tkinter` module provides a class named `IntVar` that can be used along with `Radiobutton` widgets. When you create a group of `Radiobuttons`, you associate them all with the same `IntVar` object. You also assign a unique integer value to each `Radiobutton` widget. When one of the `Radiobutton` widgets is selected, it stores its unique integer value in the `IntVar` object.

Program 12-12 demonstrates how to create and use `Radiobuttons`. Figure 12-23. shows the window that the program displays. When the user clicks the OK button an info dialog box appears indicating which of the `Radiobuttons` is selected.

Program 12-12 (radiobutton_demo.py)

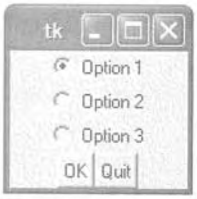
```
# This program demonstrates a group of Radiobutton widgets.

import Tkinter
import tkMessageBox

class MyGUI:
    def __init--(self):
        # Create the main window.
        self.main_window = Tkinter.Tk()

        # Create two frames. One for the Radiobuttons
        # and another for the regular Button widgets.
        self.top_frame = Tkinter.Frame(self.main_window)
        self.bottom-frame = Tkinter.Frame(self.main_window)

        # Create an IntVar object to use with
        # the Radiobuttons.
        self.radio_var = Tkinter.IntVar()

        # Set the intVar object to 1.
```

(program continues)

Program 12-12 *(continued)*

```
        self.radio_var.set(1)

        # Create the Radiobutton widgets in the top-frame.
        self.rb1 = Tkinter.Radiobutton(self.top_frame, \
                    text='Option 1', variable=self.radio_var, \
                    value=1)
        self.rb2 = Tkinter.Radiobutton(self.top_frame, \
                    text='Option 2', variable=self.radio_var, \
                    value=2)
        self.rb3 = Tkinter.Radiobutton(self.top_frame, \
                    text='Option 3', variable=self.radio_var, \
                    value=3)

        # Pack the Radiobuttons.
        self.rb1.pack()
        self.rb2.pack()
        self.rb3.pack()

        # Create an OK button and a Quit button.
        self.ok_button = Tkinter.Button(self.bottom_frame, \
                      text='OK', command=self.show_choice)
        self·quit-button = Tkinter.Button(self.bottom_frame, \
                      text='Quit', command=self.main_window.quit)

        # Pack the Buttons.
        self.ok_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.top_frame.pack()
        self.bottom_frame.pack()

        # Start the mainloop.
        Tkinter.mainloop()

    # The show-choice method is the callback function for the
    # OK button.

    def show_choice(self):
        tkMessageBox.showinfo('Selection', 'You selected option ' +\
                              str(self.radio_var.get()))

# Create an instance of the MyGUI class.
my_gui = MyGUI()
```

Figure 12-23 Window displayed by Program 12-12



Line 18 creates an `IntVar` object named `radio_var`. Line 21 calls the `radio_var` object's `set` method to store the integer value 1 in the object. (You will see the significance of this in a moment.)

Lines 24, 25, and 26 create the first `Radiobutton` widget. The argument `variable =self.radio_var` (in line 25) associates the `Radiobutton` with the `radio_var` object. The argument `value=1` (in line 26) assigns the integer 1 to this `Radiobutton`. As a result, any time this `Radiobutton` is selected, the value 1 will be stored in the `radio_var` object.

Lines 27, 28, and 29 create the second `Radiobutton` widget. Notice that this `Radiobutton` is also associated with the `radio_var` object. The argument `value=2` (in line 29) assigns the integer 2 to this `Radiobutton`. As a result, any time this `Radiobutton` is selected, the value 2 will be stored in the `radio_var` object.

Lines 30, 31, and 32 create the third `Radiobutton` widget. This `Radiobutton` is also associated with the `radio_var` object. The argument `value=3` (in line 32) assigns the integer 3 to this `Radiobutton`. As a result, any time this `Radiobutton` is selected, the value 3 will be stored in the `radio_var` object.

The `show_choice` method in lines 59 through 61 is the callback function for the OK button. When the method executes it calls the `radio_var` object's `get` method to retrieve the value stored in the object. The value is displayed in an info dialog box.

Did you notice that when the program runs the first `Radiobutton` is initially selected? This is because we set the `radio_var` object to the value 1 in line 21. Not only can the `radio_var` object be used to determine which `Radiobutton` was selected, but it can also be used to select a specific `Radiobutton`. When we store a particular `Radiobutton`'s value in the `radio_var` object, that `Radiobutton` will become selected.

Using Callback Functions with Radiobuttons

Program 12-12 waits for the user to click the OK button before it determines which `Radiobutton` was selected. If you prefer, you can also specify a callback function with `Radiobutton` widgets. Here is an example:

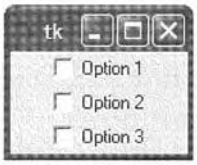
```
self.rb1 = Tkinter.Radiobutton(self.top_frame, \
        text='Option 1', variable=self.radio_var, \
        value=1, command=self.my_method)
```

This code uses the argument `command=self.my_method` to specify that `my_method` is the callback function. The method `my_method` will be executed immediately when the `Radiobutton` is selected.

Check Buttons

A *check* button appears as a small box with a label appearing next to it. The window shown in Figure 12-24 has three check buttons.

Figure 12-24 A group of check buttons



Like radio buttons, check buttons may be selected or deselected. When a check button is selected, a small check mark appears inside its box. Although check buttons are often displayed in groups, they are not used to make mutually exclusive selections. Instead, the user is allowed to select any or all of the check buttons that are displayed in a group.

You use the `Tkinter` module's `Checkbutton` class to create `Checkbutton` widgets. As with `Radiobuttons`, you can use an `IntVar` object along with a `Checkbutton` widget. Unlike `Radiobuttons`, however, you associate a different `IntVar` object with each `Checkbutton`. When a `Checkbutton` is selected, its associated `IntVar` object will hold the value 1. When a `Checkbutton` is selected, its associated `IntVar` object will hold the value 0.

Program 12-13 demonstrates how to create and use `Checkbuttons`. Figure 12-25 shows the window that the program displays. When the user clicks the OK button an info dialog box appears indicating which of the `Checkbuttons` is selected.

Program 12-13 (checkbutton_demo.py)

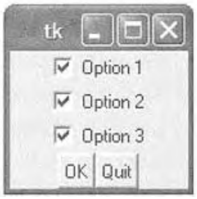
```
# This program demonstrates a group of Checkbutton widgets.

import Tkinter
import tkMessageBox

class MyGUI:
    def __init__(self):
        # Create the main window.
        self.main_window = Tkinter.Tk()

        # Create two frames. One for the checkbuttons
        # and another for the regular Button widgets.
        self.top-frame = Tkinter.Frame(self.main_window)
        self.bottom_frame = Tkinter.Frame(self.main_window)

        # Create three IntVar objects to use with
        # the Checkbuttons.
        self.cb_var1 = Tkinter.IntVar()
        self.cb_var2 = Tkinter.IntVar()
        self.cb_var3 = Tkinter.IntVar()

        # Set the intVar objects to 0.
        self.cb_var1.set(0)
        self.cb_var2.set(0)
        self.cb_var3.set(0)

        # Create the Checkbutton widgets in the top-frame.
        self.cb1 = Tkinter.Checkbutton(self.top_frame, \
                    text='Option 1', variable=self.cb_var1)
        self.cb2 = Tkinter.Checkbutton(self.top_frame, \
                    text='Option 2', variable=self.cb_var2)
        self.cb3 = Tkinter.Checkbutton(self.top_frame, \
                    text='Option 3', variable=self.cb_var3)

        # Pack the Checkbuttons.
        self.cb1.pack()
        self.cb2.pack()
        self.cb3.pack()

        # Create an OK button and a Quit button.
        self.ok_button = Tkinter.Button(self.bottom_frame, \
                      text='OK', command=self.show_choice)
        self.quit_button = Tkinter.Button(self.bottom_frame, \
                      text='Quit', command=self.main_window.quit)

        # Pack the Buttons.
        self.ok_button.pack(side='left')
        self.quit_button.pack(side='left')

        # Pack the frames.
        self.top_frame.pack()
        self.bottom_frame.pack()

        # Start the mainloop.
        Tkinter.mainloop()

    # The show-choice method is the callback function for the
```

(program continues)

Program 12-13 *(continued)*

```
    # OK button.

    def show_choice(self):
        # Create a message string.
        self.message = 'You selected:\n'

        # Determine which Checkbuttons are selected and
        # build the message string accordingly.
        if self.cb_var1.get() == 1:
            self.message = self.message + '1\n'
        if self.cb_var2.get() == 1:
            self.message = self.message + '2\n'
        if self.cb_var3.get() == 1:
            self.message = self.message + '3\n'

        # Display the message in an info dialog box.
        tkMessageBox.showinfo('Selection', self.message)

# Create an instance of the MyGUI class.
my_gui = MyGUI()
```

Figure 12-25 Window displayed by Program 12-13



Checkpoint

12.15 You want the user to be able to select only one item from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

12.16 You want the user to be able to select any number of items from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

12.17 How can you use an `IntVar` object to determine which `Radiobutton` has been selected in a group of `Radiobuttons`?

12.18 How can you use an `IntVar` object to determine whether a `Checkbutton` has been selected?

Review Questions

Multiple Choice

1. The ______________ is the part of a computer with which the user interacts.
   a. central processing unit
   b. user interface
   c. control system
   d. interactivity system
2. Before GUIs became popular, the ______________ interface was the most commonly used.
   a. command line
   b. remote terminal
   c. sensory
   d. event-driven
3. A ______________ is a small window that displays information and allows the user to perform actions.
   a. menu
   b. confirmation window
   c. startup screen
   d. dialog box
4. These types of programs that are event driven.
   a. command line
   b. text-based
   c. GUI
   d. procedural
5. An item that appears in a program's graphical user interface is known as a ______________.
   a. gadget
   b. widget
   c. tool
   d. iconified object
6. You can use this module in Python to create GUI programs.
   a. `GUI`
   b. `PythonGui`
   c. `Tkinter`
   d. `Tgui`
7. This widget is an area that displays one line of text.
   a. `Label`
   b. `Entry`
   c. `TextLine`
   d. `Canvas`
8. This widget is an area in which the user may type a single line of input from the keyboard.
   a. `Label`
   b. `Entry`

   c. `TextLine`
   d. `Input`

9. This widget is a container that can hold other widgets.
   a. `Grouper`
   b. `Composer`
   c. `Fence`
   d. `Frame`

10. This method arranges a widget in its proper position, and it makes the widget visible when the main window is displayed.
    a. `pack`
    b. `arrange`
    c. `position`
    d. `show`

11. A(n) ______________ is a function or method that is called when a specific event occurs.
    a. callback function
    b. auto function
    c. startup function
    d. exception

12. The `showinfo` function is in this module.
    a. `Tkinter`
    b. `TkInfo`
    c. `sys`
    d. `tkMessageBox`

13. You can call this method to close a GUI program.
    a. The root widget's `close` method
    b. Any widget's `quit` method
    c. The `sys.shutdown` function
    d. The `Tk.shutdown` method

14. You call this method to retrieve data from an `Entry` widget.
    a. `get_entry`
    b. `data`
    c. `get`
    d. `retrieve`

15. An object of this type can be associated with a `Label` widget, and any data stored in the object will be displayed in the `Label`.
    a. `StringVar`
    b. `LabelVar`
    c. `LabelValue`
    d. `DisplayVar`

16. If there are a group of these in a container, only one of them can be selected at any given time.
    a. `Checkbutton`
    b. `Radiobutton`
    c. `Mutualbutton`
    d. `Button`

True or False

1. The Python language has built-in keywords for creating GUI programs.
2. Every widget has a `quit` method that can be called to close the program.
3. The data that you retrieve from an `Entry` widget is always of the `int` data type.
4. A mutually exclusive relationship is automatically created among all `Radiobutton` widgets in the same container.
5. A mutually exclusive relationship is automatically created among all `Checkbutton` widgets in the same container.

Short Answer

1. When a program runs in a text-based environment, such as a command line interface, what determines the order in which things happen?
2. What does a widget's `pack` method do?
3. What does the `Tkinter` module's `mainloop` function do?
4. If you create two widgets and call their `pack` methods with no arguments, how will the widgets be arranged inside their parent widget?
5. How do you specify that a widget should be positioned as far left as possible inside its parent widget?
6. How do you retrieve data from an `Entry` widget?
7. How can you use a `StringVar` object to update the contents of a `Label` widget?
8. How can you use an `IntVar` object to determine which `Radiobutton` has been selected in a group of `Radiobuttons`?
9. How can you use an `IntVar` object to determine whether a `Checkbutton` has been selected?

Algorithm Workbench

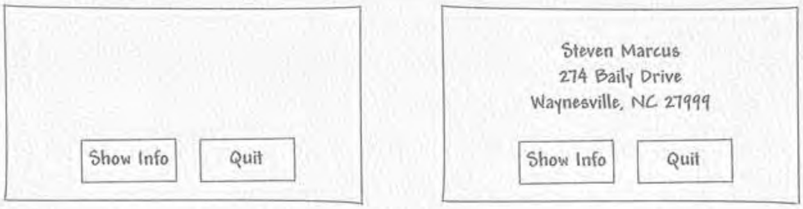
1. Write a statement that creates a `Label` widget. Its parent should be `self.main_window` and its text shoud be `'Programming is fun!'`
2. Assume `self.label1` and `self.label2` reference two `Label` widgets. Write code that packs the two widgets so they are positioned as far left as possible inside their parent widget.
3. Write a statement that creates a `Frame` widget. Its parent should be `self.main_window`.
4. Write a statement that displays an info dialog box with the title "Program Paused" and the message "Click OK when you are ready to continue."
5. Write a statement that creates a `Button` widget. Its parent should be `self.button_frame`, its text should be `'Calculate'`, and its callback function should be the `self.calculate` method.
6. Write a statement that creates a `Button` widget that closes the program when it is clicked. Its parent should be `self.button_frame`, its text should be `'Quit'`.
7. Assume the variable `data_entry` references an `Entry` widget. Write a statement that retrieves the data from the widget, converts it to an `int`, and assigns it to a variable named `var`.

Programming Exercises

1. Name and Address

Write a GUI program that that displays your name and address when a button is clicked. The program's window should appear as the sketch on the left side of Figure 12-26 when it runs. When the user clicks the *Show Info* button, the program should display your name and address, as shown in the sketch on the right of the figure.

Figure 12-26 Name and address program



2. Latin Translator

Look at the following list of Latin words and their meanings.

Latin	English
sinister	left
dexter	right
medium	center

Write a GUI program that translates the Latin words to English. The window should have three buttons, one for each Latin word. When the user clicks a button, the program displays the English translation in a label.

3. Miles-per-Gallon Calculator

Write a GUI program that calculates a car's gas mileage. The program's window should have `Entry` widgets that let the user enter the number of gallons of gas the car holds, and the number of miles it can be driven on a full tank. When a *Calculate MPG* button is clicked, the program should display the number of miles that the car may be driven per gallon of gas. Use the following formula to calculate miles-per-gallon:

$$MPG = \frac{miles}{gallons}$$

4. Celsius to Fahrenheit

Write a GUI program that converts Celsius temperatures to Fahrenheit temperatures. The user should be able to enter a Celsius temperature, click a button, and then see the equivalent Fahrenheit temperature. Use the following formula to make the conversion:

$$F = \frac{9}{5}C + 32$$

F is the Fahrenheit temperature and C is the Celsius temperature.

5. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at $10,000, its assessment value is $6,000. The property tax is then $0.64 for each $100 of the assessment value. The tax for the acre assessed at $6,000 will be $38.40. Write a GUI program that displays the assessment value and property tax when a user enters the actual value of a property.

6. Joe's Automotive

Joe's Automotive performs the following routine maintenance services:

- Oil change—$26.00
- Lube job—$18.00
- Radiator flush—$30.00
- Transmission flush—$80.00
- Inspection—$15.00
- Muffler replacement—$100.00
- Tire rotation—$20.00

Write a GUI program with check buttons that allow the user to select any or all of these services. When the user clicks a button the total charges should be displayed.

7. Long-Distance Calls

A long-distance provider charges the following rates for telephone calls:

Rate Category	Rate per Minute
Daytime (6:00 A.M. through 5:59 P.M.)	$0.07
Evening (6:00 P.M. through 11:59 P.M.)	$0.12
Off-Peak (midnight through 5:59 A.M.)	$0.05

Write a GUI application that allows the user to select a rate category (from a set of radio buttons), and enter the number of minutes of the call into an `Entry` widget. An info dialog box should display the charge for the call.

APPENDIX

A Installing Python

Before you can run Python programs on your computer you will need to install the Python interpreter. A version of Python for Windows is included on the Student CD that accompanies this book. If you can't locate the Student CD, you can download the latest version of the Python Windows installer from `www.python.org/download`. The website also provides downloadable versions of Python for several other operating systems.

Installing Python

When you execute the Python Windows installer, it's best to accept all of the default settings by clicking the Next button on each screen. (Answer "Yes" if you are prompted with any Yes/No questions.) As you perform the installation, take note of the directory where Python is being installed. It will be something similar to `C:\Python25`. (The `25` in the path name represents the Python version. At the time of this writing Python 2.5 is the most recent version.) You will need to remember this location after finishing the installation.

When the installer is finished, the Python interpreter, the IDLE programming environment, and the Python documentation will be installed on your system. When you click the Start button and look at your All Programs list you should see a program group named something like *Python* 2.5. The program group will contain the following items:

- IDLE (Python GUI)—When you click this item the IDLE programming environment will execute. IDLE is an integrated development environment that you can use to create, edit, and execute Python programs. See Appendix B for a brief introduction to IDLE.
- Module Docs—This item launches a utility program that allows you to browse documentation for the modules in the Python standard library.
- Python Command Line—Clicking this item launches the Python interpreter in interactive mode.
- Python Manuals—This item opens the Python Manuals in your web browser. The manuals include tutorials, a reference section for the Python standard library, an in-depth reference for the Python language, and information on many advanced topics.
- Uninstall Python—This item removes Python from your system.

Adding the Python Directory to the `Path` Variable

If you plan to execute the Python interpreter from a command prompt window, you will probably want to add the Python directory to the existing contents of your system's `Path` variable. (You saw the name of the Python directory while installing Python. It is something similar to `C:\Python25`.) Doing this will allow your system to find the Python interpreter from any directory when you run it at the command-line.

Use the following instructions to edit the `Path` variable under Windows XP and Windows Vista.

Windows XP

- Open the Control Panel.
- Double-click the *System* icon. (If you are running Windows XP in Category View, click *Performance and Maintenance* in the Control Panel, and then click the *System* icon.)
- Click the *Advanced* tab.
- Click the *Environment Variables* button. In the *System Variables* list, scroll to the `Path` variable.
- Select the `Path` variable and click the *Edit* button. Add a semicolon to the end of the existing contents, and then add the Python directory path.
- Click the *OK* button.

Windows Vista

- Open the Control Panel.
- Select System and Maintenance.
- Select System.
- Select Advanced System Settings.
- Click the Environment Variables button.
  In the *System Variables* list, scroll to the `Path` variable.
- Select the `Path` variable and click the *Edit* button. Add a semicolon to the end of the existing contents, and then add the Python directory path.
- Click the *OK* button.

APPENDIX

B Introduction to IDLE

IDLE is an integrated development environment that combines several development tools into one program, including the following:

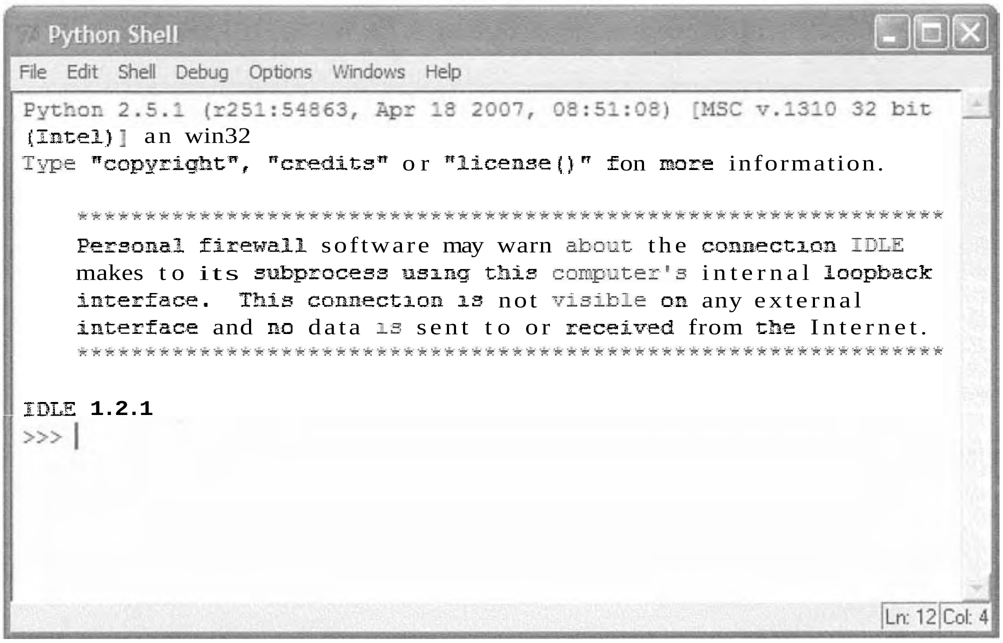
- A Python shell running in interactive mode. You can type Python statements at the shell prompt and immediately execute them. You can also run complete Python programs.
- A text editor that color codes Python keywords and other parts of programs.
- A "check module" tool that checks a Python program for syntax errors without running the program.
- Search tools that allow you to find text in one or more files.
- Text formatting tools that help you maintain consistent indentation levels in a Python program.
- A debugger that allows you to single-step through a Python program and watch the values of variables change as each statement executes.
- Several other advanced tools for developers.

The IDLE software is bundled with Python. When you install the Python interpreter, IDLE is automatically installed as well. This appendix provides a quick introduction to IDLE, and describes the basic steps of creating, saving, and executing a Python program.

Starting IDLE and Using the Python Shell

After Python is installed on your system a Python program group will appear in your Start menu's program list. One of the items in the program group will be titled IDLE *(Python GUI)*. Click this item to start IDLE and you will see the Python Shell window shown in Figure B-1. Inside this window the Python interpreter is running in interactive mode, and at the top of the window is a menu bar that provides access to all of IDLE's tools.

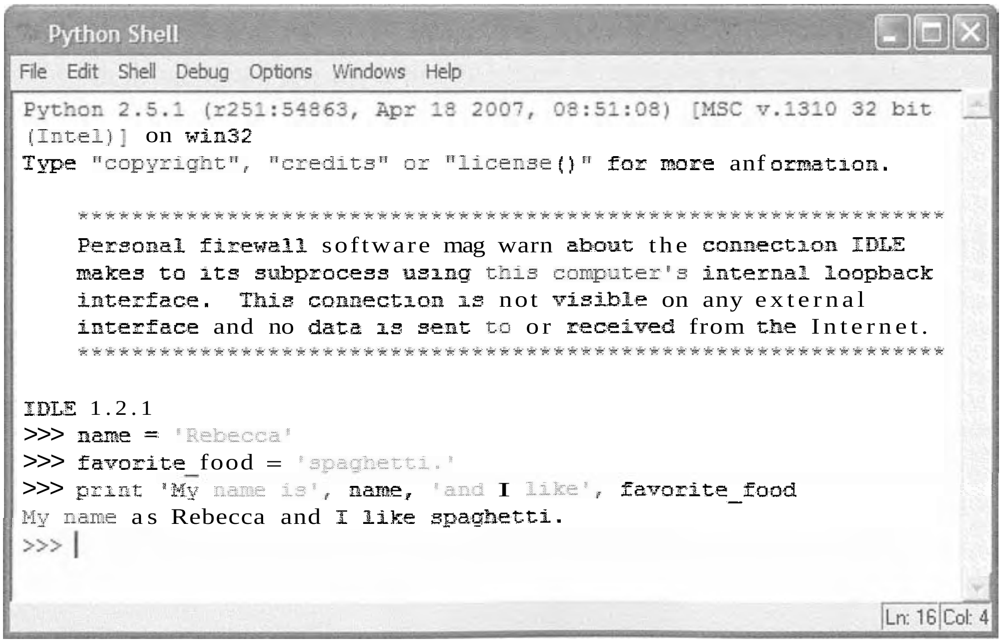
Figure B-1 IDLE shell window



```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> |
```

The >>> prompt indicates that the interpreter is waiting for you to type a Python statement. When you type a statement at the >>> prompt and press the Enter key, the statement is immediately executed. For example, Figure B-2 shows the Python Shell window after three statements have been entered and executed.

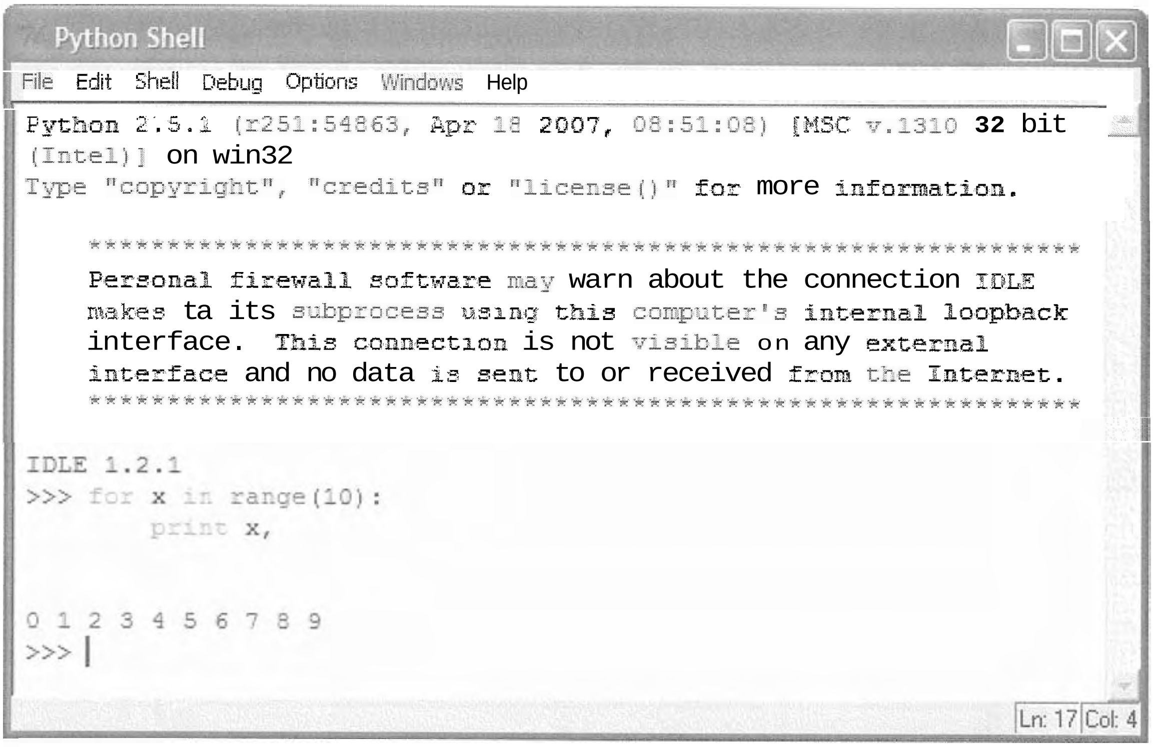
Figure B-2 Statements executed by the Python interpreter



```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> name = 'Rebecca'
>>> favorite_food = 'spaghetti.'
>>> print 'My name is', name, 'and I like', favorite_food
My name is Rebecca and I like spaghetti.
>>> |
```

When you type the beginning of a multiline statement, such as an `if` statement or a loop, each subsequent line is automatically indented. Pressing the Enter key on an empty line indicates the end of the multiline statement and causes the interpreter to execute it. Figure B-3 shows the Python Shell window after a `for` loop has been entered and executed.

Figure B-3 A multiline statement executed by the Python interpreter

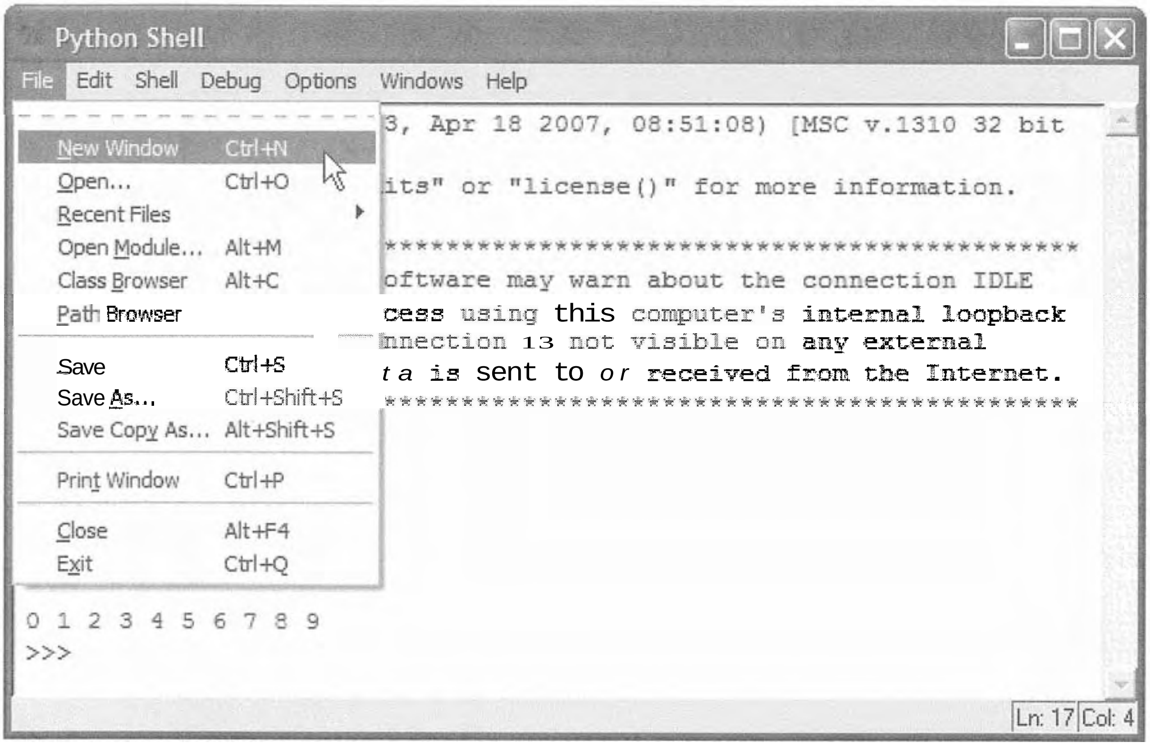


```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> for x in range(10):
	print x,

	
0 1 2 3 4 5 6 7 8 9
>>> 
```

Writing a Python Program in the IDLE Editor

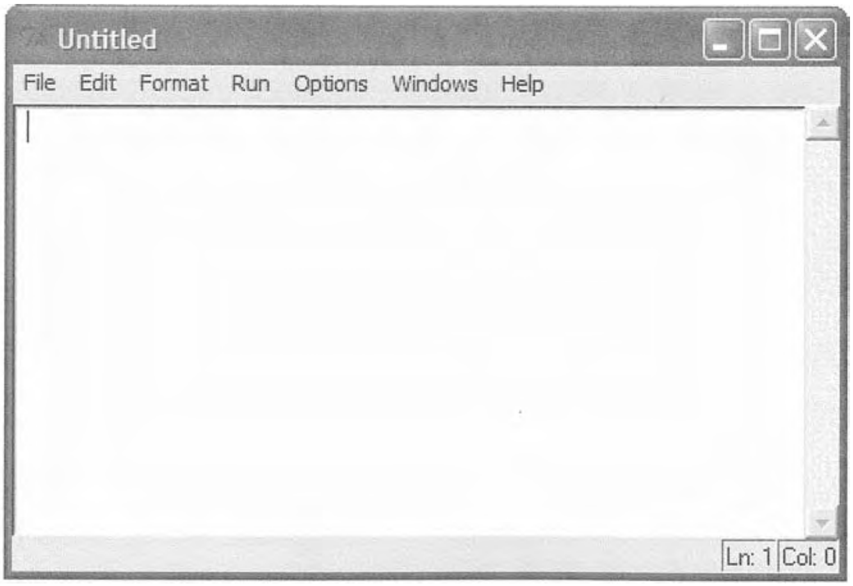
To write a new Python program in IDLE you open a new editing window. As shown in Figure B-4 you click File on the menu bar, then click New Window. (Alternatively you can press Ctrl+N.) This opens a text editing window like the one shown in Figure B-5.

Figure B-4 The File menu


Python Shell
File
Edit
Shell
Debug
Options
Windows
Help
New Window Ctrl+N
Open... Ctrl+O
Recent Files
Open Module... Alt+M
Class Browser Alt+C
Path Browser
Save Ctrl+S
Save As... Ctrl+Shift+S
Save Copy As... Alt+Shift+S
Print Window Ctrl+P
Close Alt+F4
Exit Ctrl+Q
Ln: 17
Col: 4

```
3, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit

its" or "license()" for more information.

**********************************************
oftware may warn about the connection IDLE
cess using this computer's internal loopback
nnection is not visible on any external
ta is sent to or received from the Internet.
**********************************************

0 1 2 3 4 5 6 7 8 9
>>>
```

Figure B-5 A text editing window


Untitled
File
Edit
Format
Run
Options
Windows
Help
Ln: 1
Col: 0

To open a program that already exists, click File on the menu bar, then Open. Simply browse to the file's location and select it, and it will be opened in an editor window.

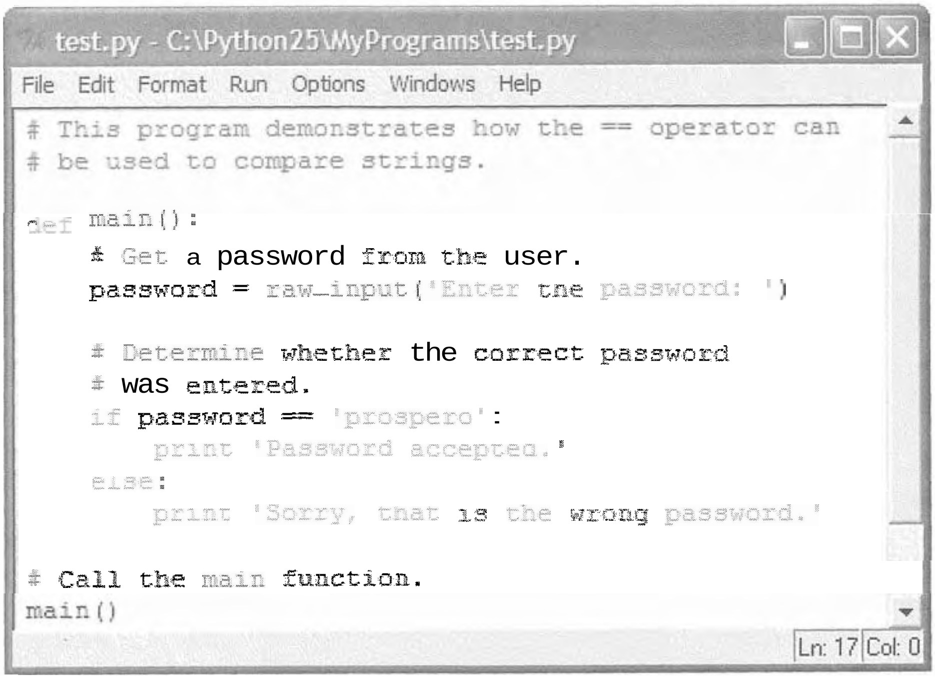
Color Coding

Code that is typed into the editor window, as well as in the Python Shell window, is colorized as follows:

- Python keywords are displayed in orange.
- Comments are displayed in red.
- String literals are displayed in green.
- Defined names, such as the names of functions and classes, are displayed in blue.
- Built-in functions are displayed in purple.

Figure B-6 shows an example of the editing window containing colorized Python code.

Figure B-6 Colorized code in the editing window



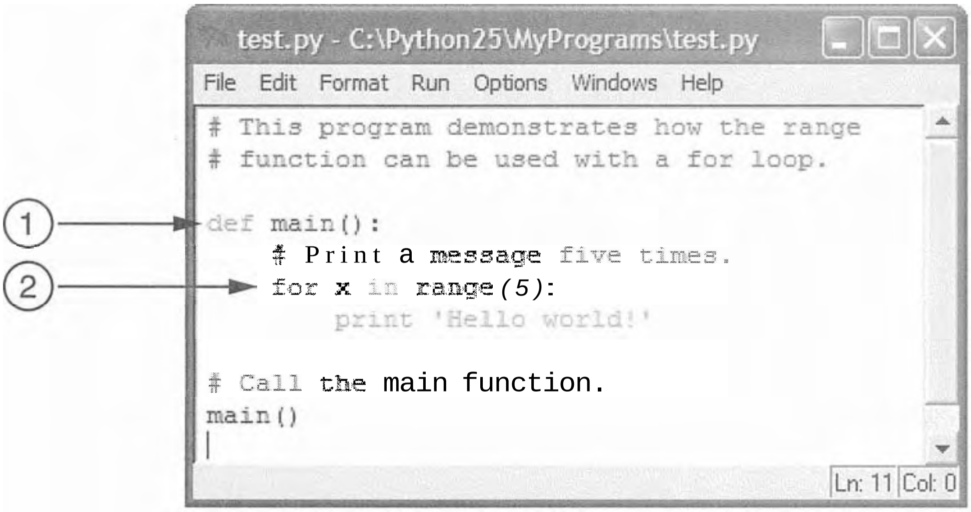
```
# This program demonstrates how the == operator can
# be used to compare strings.

def main():
    # Get a password from the user.
    password = raw_input('Enter the password: ')

    # Determine whether the correct password
    # was entered.
    if password == 'prospero':
        print 'Password accepted.'
    else:
        print 'Sorry, that is the wrong password.'

# Call the main function.
main()
```

TIP: You can change IDLE's color settings by clicking Options on the menu bar, then clicking Configure IDLE. Select the Highlighting tab at the top of the dialog box, and you can specify colors for each element of a Python program.

Automatic Indentation

The IDLE editor has features that help you to maintain consistent indentation in your Python programs. Perhaps the most helpful of these features is automatic indentation. When you type a line that ends with a colon, such as an `if` clause, the first line of a loop, or a function header, and then press the Enter key, the editor automatically indents the lines that are entered next. For example, suppose you are typing the code shown in Figure B-7. After you press the Enter key at the end of the line marked ①, the editor will automatically indent the lines that you type next. Then, after you press the Enter key at the end of the line marked ②, the editor indents again. Pressing the Backspace key at the beginning of an indented line cancels one level of indentation.

Figure B-7 Lines that cause automatic indentation


test.py - C:\Python25\MyPrograms\test.py
File Edit Format Run Options Windows Help
①
②
Ln: 11 Col: 0

```
# This program demonstrates how the range
# function can be used with a for loop.

def main():
    # Print a message five times.
    for x in range(5):
        print 'Hello world!'

# Call the main function.
main()
```

By default, IDLE indents four spaces for each level of indentation. It is possible to change the number of spaces by clicking Options on the menu bar, then clicking Configure IDLE. Make sure Fonts/Tabs is selected at the top of the dialog box, and you will see a slider bar that allows you to change the number of spaces used for indentation width. However, because four spaces is the standard width for indentation in Python, it is recommended that you keep this setting.

Saving a Program

In the editor window you can save the current program by performing any of these operations from the File menu:

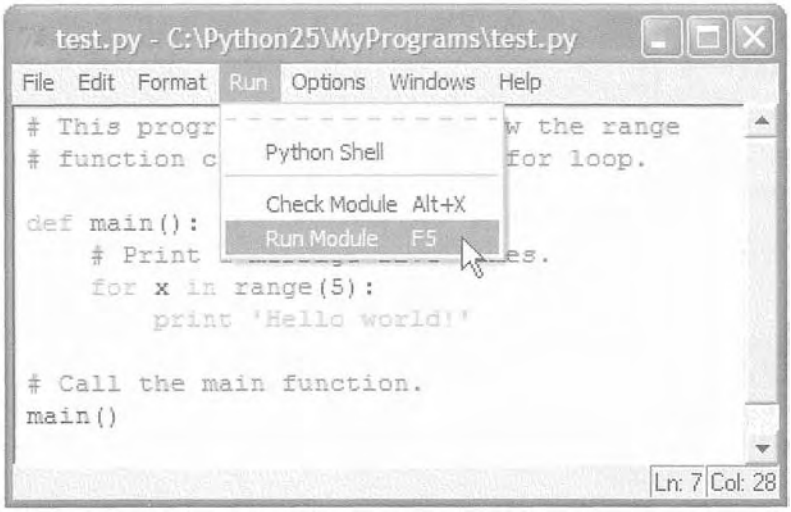
- Save
- Save As
- Save Copy As

The Save and Save As operations work just as they do in any Windows application. The Save Copy As operation works like Save As, but it leaves the original program in the editor window.

Running a Program

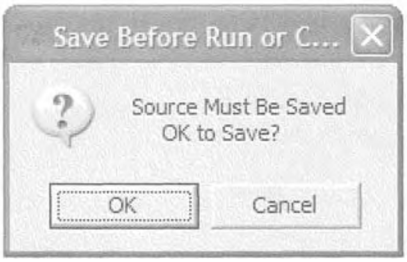
Once you have typed a program into the editor, you can run it by pressing the F5 key, or as shown in Figure B-8, by clicking Run on the editor window's menu bar, then Run Module. If the program has not been saved since the last modification was made, you will see the dialog box shown in Figure B-9. Click OK to save the program. When the program runs you will see its output displayed in IDLE's Python Shell window, as shown in Figure B-10.

Figure B-8 The editor window's Run menu


test.py - C:\Python25\MyPrograms\test.py
File
Edit
Format
Run
Options
Windows
Help
Python Shell
Check Module Alt+X
Run Module F5
Ln: 7
Col: 28

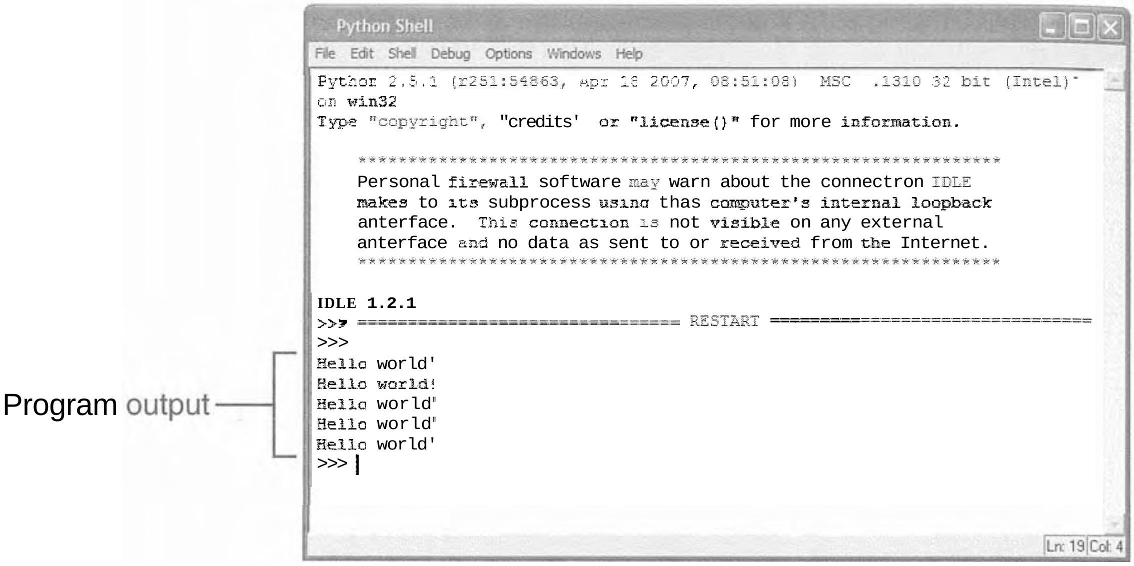
```
# This progr          w the range
# function c          for loop.

def main():
    # Print                    es.
    for x in range(5):
        print 'Hello world!'

# Call the main function.
main()
```

Figure B-9 Save confirmation dialog box


Save Before Run or C...
OK
Cancel

Source Must Be Saved
OK to Save?

Figure B-10 Output displayed in the Python Shell window

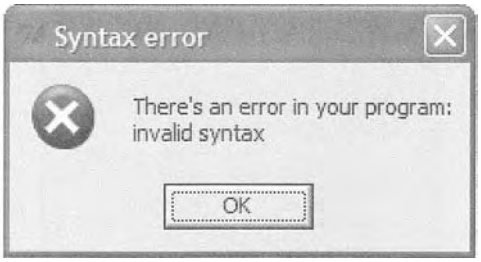


```
Python 2.5.1 (r251:54863, apr 18 2007, 08:51:08)  MSC  .1310 32 bit (Intel)
on win32
Type "copyright", "credits'  or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connectron IDLE
    makes to its subprocess using thas computer's internal loopback
    anterface.  This connection is not visible on any external
    anterface and no data as sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> ================================ RESTART ================================
>>>
Hello world'
Hello world!
Hello world"
Hello world"
Hello world'
>>>
```

Program output

If a program contains a syntax error, when you run the program you will see the dialog box shown in Figure B-11. After you click the OK button the editor will highlight the location of the error in the code. If you want to check the syntax of a program without trying to run it, you can click Run on the menu bar, then Check Module. Any syntax errors that are found will be reported.

Figure B-11 Dialog box reporting a syntax error



There's an error in your program:
invalid syntax

Other Resources

This appendix has provided an overview for using IDLE to create, save, and execute programs. IDLE provides many more advanced features. To read about additional capabilities, see the official IDLE documentation at www.python.org/idle.

APPENDIX

C The ASCII Character Set

The following table lists the ASCII (American Standard Code for Information Interchange) character set, which is the same as the first 127 Unicode character codes. This group of character codes is known as the *Latin Subset* of *Unicode*. The code columns show character codes and the character columns show the corresponding characters. For example, the code *65* represents the letter A. Note that the first 31 codes, and code 127, represent control characters that are not printable.

Code	Character	Code	Character	Code	Character	Code	Character	Code	Character
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	Escape	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32	(*Space*)	58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	Backspace	34	“	60	<	86	V	112	p
9	HTab	35	#	61	=	87	W	113	q
10	Line Feed	36	$	62	>	88	X	114	r
11	VTab	37	%	63	?	89	Y	115	s
12	Form Feed	38	&	64	@	90	Z	116	t
13	CR	39		65	A	91	[	117	u
14	SO	40	(	66	B	92	\	118	v
15	SI	41	)	67	C	93	]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	Y
18	DC2	44		70	F	96		122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46		72	H	98	b	124	|
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	DEL
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

Index

Symbols

- # character, 39
- % operator, 175
- *= operator, 175
- * operator
  - multiplication, 53
  - repetition, 292–293
- += operator, 175
- + operator
  - for addition, 53
  - for string concatenation, 63–64
- <= operator
  - defined, 116
  - relationship testing, 117
- < operator
  - defined, 116
  - in string comparisons, 129
- == operator
  - assignment operator versus, 117
  - defined, 116
  - in string comparisons, 126
  - use example, 119
  - value comparison, 117
- = (assignment operator), 41
- -= operator, 175
- /= operator, 175
- != operator
  - defined, 116
  - example use, 119
  - value determination, 117
- / (division operator), 53
- >= operator
  - defined, 116
  - relationship testing, 117
- >>> prompt, 21, 22, 460
- > operator
  - defined, 116
  - example use, 117
  - flowchart, 118
  - in string comparisons, 128–129
- / separator, 321

A

- Accessor methods, 355
- Accumulators, 173, 266
- `acos()` function, 213
- Actions
  - class, 365
  - conditionally executed, 114
- Ada programming language, 17
- Addition (+) operator
  - in concatenating lists, 307
  - precedence, 53
- Algebraic expressions
  - conversion example, 57–58
  - converting to programming statements, 56–58
  - examples, 56
- Algorithms
  - defined, 33
  - example, 33
  - recursive, 406–413
- `and` operator
  - defined, 138
  - short-circuit evaluation, 140
  - truth table, 139
  - use example, 139
- `append()` method
  - defined, 300, 301
  - first call, 302
  - program example, 301–302
- Append mode, 238
- Appending, data to files, 238
- Application software, 7
- Arguments
  - defined, 93
  - keyword, 101–103
  - listing order, 98–99
  - multiple, passing, 97–99
  - parameter changes and, 99–101
  - passed to `range` function, 165
  - passing lists as, 311–312
  - passing to functions, 93–103
  - passing variables as, 95
  - positional, mixing with keyword arguments, 103
  - as step values, 165
- Arrays, 295
- ASCII (American Standard Code for Information Interchange) code, 11–12, 127
- ASCII character set, 467
- `asin()` function, 213
- Assemblers, 16
- Assembly language, 15–16
- Assignment operator (=)
  - augmented, 175–176
  - defined, 41
  - equality operator (==) versus, 117
- Assignment statements
  - example, 40
  - format, 41
  - global variable creation, 104
  - math expression, 50–51
  - moving, 92
  - variable creation with, 40–43
- `atan()` function, 213
- `AttributeError` exception, 392
- Augmented assignment operators
  - defined, 175
  - list of, 175
  - using, 176

- `Automobile` class
  - classes inheriting from, 377–378
  - code, 375–376
  - as complete class, 377
  - general data, 377
  - `get— doors` method, 379
  - `get—make` method, 379
  - `get_mileage` method, 379
  - `get_model` method, 379
  - `get— price` method, 379
  - `__init__` method, 376, 378
  - `set— doors` method, 378
  - subclasses inheriting from, 377
  - superclass, 375
  - in UML diagram, 383
- Averages
  - calculating, 55–56
  - list value, 310–311

B

- Backslash (\). *See* Escape characters
- `BankAccount` class example
  - `get_balance` method, 347
  - `__init__` method, 345, 347
  - object creation, 345
  - program, 345
  - use program, 346–347
  - `withdraw` method, 347
- Base case. *See* also Recursion
  - defined, 403
  - Towers of Hanoi algorithm, 412
- Base classes, 374. *See* also Superclasses
- BASIC, 17
- Binary digits
  - values, 9–10
  - values, determining, 10
- Binary files, 227
- Binary numbering system, 9
- Bits
  - all set to 0, 11
  - all set to 1, 11
  - defined, 8
  - patterns, 9–10
- Blocks
  - blank lines in, 85
  - defined, 80
  - last line in, 84
  - line indentations, 84–85
  - nested, 120–121, 134
- `bool` data type, 144
- Boolean expressions
  - compound, 138, 143
  - defined, 115–116
  - with logical operators, 138
  - with relational operators, 116
  - tested by `if` statements, 116
- Boolean functions
  - defined, 209
  - in testing conditions, 209
  - in validation code, 210
- Boolean variables
  - defined, 144
  - example use, 145
  - `False`, 144
  - as flags, 145
  - `True`, 144
- Buffers, 230
- `Button` widgets, 430
- Buttons
  - Convert, 434
  - Quit, 432–434
  - text, 430
  - use click, 432
- Bytes
  - data storage in, 9
  - defined, 8
  - for large numbers, 11

C

- C# programming language, 17
- Calculations
  - averages, 55–56
  - data type conversion, 58–60
  - evaluation rules, 58–59
  - formula conversions, 56–58
  - grouping with parentheses, 54
  - integer division, 52–53
  - math expressions, 50–51
  - math formula conversion, 56–58
  - math operators, 50, 53, 56
  - operator precedence, 53–54
  - percentages, 51–52
  - performing, 50–62
  - rounding, 58
  - running total, 173–176
- Callback functions
  - defined, 430
  - as event handlers, 430
  - with `Radiobuttons`, 447–448
- Calling functions. *See* also Functions
  - defined, 81
  - examples, 83, 87–90
  - illustrated, 82
  - in loops, 160–161
- CamelCase variable names, 43–44
- `Car` class, 366
  - defined, 375
  - inheriting from `Automobile` class, 377
  - in UML diagram, 383
- `Cat` class, 390
- C/C++ programming languages, 17
- CD drives, 6
- CDs (compact discs), 6
- `ceil()` function, 213
- `CellPhone` class example
  - attributes, 353
  - creating, 353–355
  - importing, 354
  - methods, 353
  - object storage in lists, 356–358
  - programs, 353–355
  - UML diagram, 359
- Central processing unit (CPU)
  - defined, 3
  - as electronic device, 13
  - instruction set, 14
  - microprocessors, 3, 4
  - operations, 13
  - program instructions, understanding, 13–14
- Characters
  - comment, 39
  - escape, 62–63
  - line continuation, 61
  - newline, 62
  - storing, 11–12
  - string, accessing, 274–278
  - string, comparing, 128–129
  - string, extracting, 281–283
  - string, validating, 289–292
- Check buttons
  - defined, 444, 448
  - illustrated, 448
  - selection, 448
- `Checkbutton` class, 448
- Class definitions
  - defined, 334
  - headers, 336
  - organization, 343
  - program example, 335
- Class responsibilities
  - defined, 365
  - identifying, 364–367
  - problem domain and, 365

- Classes. *See* also Inheritance; Instances
  - actions, 365
  - `Automobile`, 375–381
  - `BankAccount`, 345–347
  - `Car`, 366,375, 377–378
  - `Cat`, 390
  - `CellPhone`, 353–355
  - `Checkbutton`, 448
  - `Coin`, 337–343
  - `Customer`, 365
  - defined, 333
  - definition example, 334
  - design techniques, 358–367
  - `Dog`, 389–390
  - finding, 359–364
  - `IntVar`, 445
  - `Mammal`, 388–394
  - methods, 336
  - names, 335
  - objects versus, 333
  - `RadioButton`, 444–448
  - `ServiceQuote`, 366–367
  - storing in modules, 343–345
  - `StringVar`, 436
  - subclass, 374, 377, 388
  - superclass, 374, 388
  - `SUV`, 375, 380–381
  - `Truck`, 375, 379–380
  - UML diagrams, 358–359
- `close` method, 231
- COBOL, 17
- Code reuse
  - as function benefit, 79
  - modules in, 214
- `Coin` class example
  - definition, 343
  - expression actions, 339
  - `get_sideup` method, 337, 339,352
  - `my_coin` object, 339–340
  - program example, 337–338, 341–343
  - `sideup` attribute, 340, 341, 352
  - `toss` method, 337, 341,352
  - UML diagram, 359
- Command line interface. *See* also User interfaces
  - defined, 419
  - difficulty of use, 420
  - illustrated, 419
- Comments
  - # character, 39
  - defined, 39
  - end-line, 39–40
  - writing resistance, 40
- Comparisons, string
  - == operator in, 126
  - case-insensitive, 127
  - case-sensitive, 127, 287
  - character, 128–129
  - example, 126
  - greater than/less than, 127
  - relational operators, 128
- Compilers, 19
- Compound Boolean operators, 138, 143
- Computers
  - component illustration, 3
  - components, 2
  - CPU, 3–4
  - data storage, 8–13
  - ENIAC, 3–4
  - function of, 1
  - input devices, 6
  - main memory, 4–5
  - output devices, 6
  - secondary storage, 5–6
  - use of, 1
  - user interfaces, 419
- Concatenating lists, 307
- Condition-controlled loops. *See* also Loops
  - beginning of, 154
  - conditions tested by, 153, 155
  - defined, 152
  - flowchart, 153, 156
  - function calls, 160–161
  - infinite, 159
  - logic, 153
  - parts, 153
  - as pretest loop, 156–157
  - program design with, 157–159
  - `while` statement, 152–161
- Conditions
  - series, testing, 134–136
  - testing with Boolean functions, 209
- Constants, global, 105–107
- Control structures, 113
- Convert button, 434
- `convert` method, 436
- Cookies, 226
- Copying
  - lists, 307–308
  - records, 257, 260
- `cos()` function, 213
- Count-controlled loops. *See* also Loops
  - defined, 152, 161
  - designing, 168–170
  - examples, 163–164
  - first iteration, 162, 166
  - `for` statements, 152, 161–172
  - functioning, 163
  - iterations, 162–163
  - list generation, 172–173
  - `range` function with, 164–166
  - target variables, 163, 166–168
  - user control, 170–172
  - uses, 161–162
- `Customer` class, 365
- Customers, programmer interview with, 33

D

- Data
  - appending to existing files, 238
  - digital, 12
  - numeric, writing/reading, 238–241
  - output, 62–69
  - reading from files, 227, 232–235
  - writing to files, 226, 230–232
- Data attributes
  - defined, 330
  - hiding, 340–343
  - manipulation, 332
  - private, 355
  - public method access, 355
  - values, 332
  - values, changing, 332
  - values, retrieving with accessor methods, 355
- Data hiding, 330
- Data storage, 8–13
  - characters, 11–12
  - music, 13
  - numbers, 9–11
  - numbers, advanced, 12
  - pictures, 12

- Data types
  - `bool`, 144
  - conversion, 58–60
  - defined, 46
  - `float`, 46
  - `int`, 46
  - numeric, 45–46
  - `str`, 46–47
- Decision structures
  - conditional execution, 114
  - defined, 114
  - dual alternative, 121
  - examples, 118
  - flow chart, 114
  - nested, 130–138
  - nested blocks, 120–121
  - sequence structures with, 130–131
  - single alternative, 114
  - string comparison, 126–130
  - three actions, 115
- `degrees()` function, 213
- `del` statement, 306
- Depth of recursion, 402
- Derived classes. *See* Subclasses
- Dialog boxes. *See* also Graphical user interface (GUI)
  - defined, 420
  - illustrated, 420
  - info, 430, 431
- Digital data, 12
- Direct access files, 228
- Direct recursion, 406
- Disk drives
  - defined, 5
  - for program storage, 14
- `display_list` function, 358
- Divide and conquer, 78
- Division
  - integer, 52–53
  - truncation, 53
- Division (/) operator, 53
- Document types, 225
- Documentation exceptions, 268
- Dog class, 389–390
- Dot notation, 193
- Double-quotes, 37–38
- Dual alternative decision structure
  - defined, 121
  - flowchart, 122
  - writing, 122
- DVD drives, 6
- DVDs (digital versatile discs), 6

E

- Encapsulation, 330
- End of file
  - detection, 243–245
  - detection logic, 244
- End users, 49
- End-line comments, 39–40
- `endswith()` method, 288
- ENIAC computer, 3–4
- `Entry` widgets
  - defined, 433
  - `get` method, 434
  - program example, 434–436
- Error traps. *See* Input validation loops
- Escape characters
  - backslash (\\), 62
  - defined, 62
  - newline (\n), 232–238
  - types of, 63
- Event handlers. *See* Callback functions
- Event-driven programs, 421
- Exception handlers, 264
- Exceptions
  - `AttributeError`, 392
  - avoidance example, 263
  - defined, 262
  - experimentation, 268
  - `IndexError`, 277–278, 296,299
  - `IOError`, 263,266,267, 268
  - multiple, handling, 266–268
  - name, 263
  - preventing, 263
  - resources, 268
  - traceback, 263
  - `ValueError`, 267,268
  - `ZeroDivisionError`, 268
- Execution
  - `if` statement, 115
  - pausing, 90
- Exercises
  - classes and OOP programming, 370–372
  - computers and programming introduction, 28
  - decision structures, 148–150
  - files and exceptions, 272
  - functions, 110–111
  - GUI programming, 454–455
  - inheritance, 396–397
  - input, processing, and output, 73–75
  - repetition structures, 188–189
  - strings and lists, 326–328
  - value-returning functions and modules, 221–223
- `exp()` function, 213
- Exponent (`**`) operator
  - defined, 56
  - precedence, 53
  - using, 56

F

- `factorial` function, 404–405
- Factorials
  - calculating, 403–405
  - definition rules, 403
- `False` variable, 144
- Fetch-decode-execute cycle, 15
- Fibonacci numbers, 407
- Fibonacci series, 407–409
  - defined, 407–408
  - recursive function, 408
- Field widths
  - benefits, 67
  - minimum, specifying, 66–67
- Fields, 250
- File objects
  - defined, 228
  - methods, 230
  - variable name reference, 229
- Filename extensions, 228
- Files
  - access methods, 227–228
  - appending data to, 238
  - binary, 227
  - closing, 227, 230
  - cookie, 226
  - direct access, 228
  - document, 225
  - end of, detecting, 243–245
  - game data, 225–226
  - image, 225
  - input, 226
  - lists and, 317–320
  - with loops, reading, 243–245
  - modes, 229
  - opening, 227,229–230
  - output, 226
  - processing, 227
  - processing with loops, 242–249
  - reading data from, 227, 232–235

  - sequential access, 227–228
  - software packages storing data in, 225–226
  - spreadsheet, 225
  - temporary, 257, 260
  - text, 227
  - types of, 227
  - use steps, 227
  - working with, 247–249
  - writing data to, 226, 230–232
- `find()` method, 288
- Flags, 145
- Flash drives, 6
- Flash memory, 6
- `float()` function, 59–60, 240,267
- `float` data type. *See* also Data types
  - defined, 46
  - in mixed-type expressions, 59
  - operations on, 58
- Floating-point numbers, 64
- `floor()` function, 213
- Floppy disk drives, 5
- Flowcharts
  - decision structure, 114
  - defined, 34
  - diamond symbol, 114
  - dual alternative decision structure, 122
  - function, 85–86
  - function call in loops, 161
  - function call symbol, 85
  - illustrated, 35, 86
  - input symbols, 34
  - input validation loop, 180
  - nested decision structure, 132
  - nested loop, 185
  - output symbols, 34
  - processing symbols, 34
  - running total calculation, 173
  - sequence structures nested in decision structure, 131
  - sequence structures with decision structure, 130
  - symbols, 34
  - terminal symbols, 34
  - `while` loops, 153
- `for` loops
  - defined, 161
  - designing, 168–170
  - examples, 163–164
  - first iteration, 162, 166
  - functioning, 163
  - iterating over lists with, 296
  - iterating through lists with, 318
  - iterations, 162–163
  - list generation, 172–173
  - `range` function with, 164–166
  - to read lines, 245–246
  - for string character access, 274–276
  - target variables, 163, 166–168
  - user control, 170–172
  - uses, 161–162
- `for` statements
  - in count-controlled loops, 152
  - designing count-controlled loops with, 168–170
  - execution, 162
  - general format, 162
- Formatting
  - integers, 67–69
  - multiple values, 66
  - numbers, 64–69
  - strings, 67–69
- Formatting specifiers
  - defined, 64–65
  - minimum field width, 66–67
  - multiple values, 66
  - same number as values, 66
  - single value, 65
- FORTRAN, 17
- Frame
  - defined, 427, 428
  - organizing widgets with, 427–430
  - `pack` method, 429
  - placing widgets in, 428
  - program example, 428–429
- Frames, widget organization with, 441
- Function calls
  - in loops, 160–161
  - symbol, 85
- Function definitions
  - block, 80
  - defined, 79
  - example, 87–88
  - function header, 80
  - general format, 80
  - multiple, 82
  - writing, 80–81
- Function headers, 80
- Functions
  - `acos()`, 213
  - `asin()`, 213
  - `atan()`, 213
  - benefits, 78–79
  - Boolean, 209–210
  - built-in, 192
  - callback, 430
  - calling, 81–84, 101
  - `ceil()`, 213
  - `cos()`, 213
  - defined, 48, 77
  - `degrees()`, 213
  - `display_list`, 358
  - for divide and conquer, 78
  - `exp()`, 213
  - `factorial`, 404–405
  - `float()`, 59–60,240,267
  - `floor()`, 213
  - flowcharts, 85–86
  - global variables, 105
  - `hypot()`, 213
  - indentation, 84–85
  - `input`, 48–49
  - `int()`, 60,240,241
  - introduction to, 77–79
  - `isinstance`, 391–394
  - `len`, 278,297
  - library, 192–193
  - local variables, 91–93
  - `log10()`, 213
  - `log()`, 213
  - `make_list`, 358
  - `math` module, 211–212,213
  - `max`, 306–307
  - `message`, 400–401
  - method, 230
  - `min`, 306–307,315
  - modularizing with, 205–208
  - names, 80
  - `open`, 229–230
  - passing arguments to, 93–103
  - program control transfer to, 84
  - program design for using, 85–90
  - `radians()`, 213
  - `randint`, 193–195,335
  - `random`, 199
  - `randrange`, 198–199
  - `range`, 164–166
  - `raw_input`, 48, 49
  - recursive, 399–402
  - `rename`, 258
  - returning lists from, 312–314

Functions (continued)
  returns, 81, 84
  `showinfo`, 430,432
  simple, 191
  `sin()`, 213
  `sqrt`, 212,213
  standard library of, 192
  storing in modules, 214–218
  `str`, 238–239
  `sum-range`, 406–407
  `tan()`, 213
  `uniform`, 199
  value-returning, 191–211

G

Game data files, 225–226
Generalization, 373
GIGO, 179
Global constants
  defined, 105
  using, 105–107
  values, 105
`global` key word, 105
Global variables. *See also* Variables
  access, 103
  defined, 103
  drawbacks, 104–105
  examples, 103–104
  functions using, 105
  program understanding and, 105
  use, restricting, 104
  using, 105–107
  values, assigning, 104
Graphical user interface (GUI). *See also* GUI programs
  defined, 419, 420
  dialog boxes, 420
  as event-driven, 421
  graphical elements, 420
  libraries, 421
  programming, 419–450
Greatest common divisor (GCD)
  determination, 409
  program example, 409–410
Grouping, with parentheses, 54
GUI programs. *See also* Graphical user interface (GUI)
  buttons, 430–433
  check buttons, 448–450
  creating, 440–444
  creating with `Tkinter` module, 421–424
  as event-driven, 421
  exiting, 422
  illustrated, 421
  info dialog boxes, 430–433
  input, 433–436
  main window display, 425
  output fields, 436–440
  radio buttons, 444–448
  text display, 424–427
  widgets, 422

H

Hardware, 2
Hiding attributes, 340–343
Hierarchy charts
  defined, 87
  `if-else` statements, 124
  illustrated, 87, 88
  passing arguments to functions, 96
High-level languages, 16–17
`hypot()` function, 213

I

IDLE, 459–466
  automatic indentation, 463–464
  color coding, 463
  defined, 23, 459
  File menu, 462
  illustrated, 23
  installation, 459
  introduction to, 459–466
  resources, 466
  running, 23
  running programs from, 465–466
  saving programs, 464
  shell window, 460
  starting, 459
  Syntax error dialog box, 466
  text editing window, 462
  text editor, 23
  writing Python programs in, 461–462
`if` statement
  with `and` operator, 139
  Boolean expressions, 115–116
  defined, 113
  execution, 115
  general format, 115
  with `not` operator, 140
  with `or` operator, 139
  relational operators, 116–117
  uses, 115
`if-elif-else` statement. *See also* Nested decision structures
  alignment, 137
  defined, 136
  example, 137
  general format, 136
  indentation, 137
  logic, 137
`if-else` statements
  clause alignment, 123, 134
  condition, testing, 122
  conditional execution, 122
  defined, 121
  general format, 122
  hierarchy chart, 124
  indentation, 123–125
  nested, 133–134, 137, 141
  tests, 133
  using, 123–125
`if-then` statements, 119–120
Images
  as files, 225
  storage, 12
`import` statements
  defined, 192
  in importing modules, 215
  writing, 193
`in` operator
  finding list items with, 297–298
  testing strings with, 283
Indentation
  automatic, 85
  in blocks, 84
  as four spaces, 85
  IDLE editor, 463–464
  `if-elif-else` statement, 137
  `if-else` statement, 123–125
  line, methods, 85
`index()` method
  calling, 303–304
  defined, 301
  passing arguments to, 302
  program example, 302–303
`IndexError` exceptions
  defined, 277
  examples, 277–278
  with lists, 296, 299

- Indexes
  - defined, 276
  - invalid, 280
  - list, 296–297
  - negative, 296
  - negative numbers in, 277, 280
  - string, 276
  - string length as, 280
  - use examples, 277
- Indirect recursion, 406
- Infinite loops, 159
- Info dialog boxes. *See* also Dialog boxes
  - defined, 430
  - illustrated, 431, 433, 436
- Inheritance. *See* also Classes
  - defined, 373
  - "is a" relationships and, 374–382
  - subclass, 374
  - superclass, 374
  - in UML diagrams, 382–387
  - using, 383–387
- Initializer methods, 336
- `__init__` method, 336,345, 347,376,378
- Input
  - defined, 6
  - flowchart symbols, 34
  - program, 35–36
  - reading from keyboard, 48–50
- Input devices, 6
- Input files, 226
- `input` function
  - defined, 48
  - general format, 48
  - reading numbers with, 48–49
  - sample use, 48
- Input validation, 179, 180
- Input validation loops
  - defined, 179
  - error message display, 180
  - as error trap, 181
  - flowchart, 180
  - logic, 180
  - priming read, 180
  - writing, 181–184
- `insert()` method
  - defined, 301
  - passing arguments to, 304
  - program example, 304
- Instances. *See* also Classes
  - attributes, 350
  - creation in memory, 336
  - defined, 333
  - determining, 392
  - working with, 350–352
- `int()` function, 60
  - defined, 240
  - `readline` method argument, 241
- `int` data type
  - defined, 46
  - in mixed-type expressions, 59
  - operations on, 58
- Integer division
  - defined, 52
  - `float` function and, 60
  - truncation, 53
- Integers, 45
  - formatting, 67–69
  - list, 295
  - string conversion to, 241
- Integrated development environment, 23
- Interactive mode
  - defined, 21
  - error messages, 22
  - interpreter start in, 21
  - using, 21–22
- Interpreters
  - defined, 19
  - Python, 20, 21
- `IntVar` class, 445, 448
- `IOError` exception, 263, 266, 267,268
- IPO charts
  - defined, 204
  - descriptions, 205
  - illustrated, 204
  - use decision, 205
- "Is a" relationship
  - defined, 374
  - examples, 374
  - inheritance and, 374–382
- `isalnum()` method, 285
- `isalpha()` method, 285
- `isdigit()` method, 285
- `isinstance` function, 391–394
  - calling, 394
  - general format, 392
  - in instance determination, 392
- `islower()` method, 285
- `isspace()` method, 285
- `isupper()` method, 285
- Iterating over lists, 296
- Iterating over strings
  - with `for` loop, 274–276
  - general format, 274
  - illustrated, 275
  - program example, 276

J

- Java, 17
- JavaScript, 17

K

- Key words
  - defined, 18
  - function names and, 80
  - variable names and, 43
- Keyboard, reading input from, 48–50
- Keyword arguments. *See* also Arguments
  - arguments passed to, 101
  - defined, 101
  - example use, 101–102
  - mixing with positional arguments, 103
  - order, 102

L

- `Label` widget
  - creating, 424
  - defined, 424
  - `pack` method, 425,427
  - program examples, 424–427
  - `StringVar` object with, 436–440
- `len` function, 278
  - defined, 278
  - with lists, 297
- Libraries, 421
- Library functions
  - as black boxes, 192
  - built-in, 192
  - defined, 192
  - modules, 192
  - `randint`, 193–195
  - `random`, 199
  - for random number generation, 193
  - `randrange`, 198–199
  - `uniform`, 199
- Line continuation character ('), 63
- Lists, 294–322
  - concatenating, 307
  - copying, 307–308
  - creating with repetition (*) function, 295

Lists (continued)
  creation example, 294
  defined, 294
  displaying, 295
  as dynamic data structures, 294
  files and, 317–320
  generating, highest to lowest, 172
  generating with `range` function, 295
  indexing, 296–297
  integers, 295
  item range, summing, 406–407
  items, adding, 300–302
  items, determining, 302–304
  items, finding, 297–298
  items, in math expression, 308–309
  items, inserting, 304
  items, rearranging, 304–305
  items, removing, 305–306
  items, reversing order, 306
  iterating over, 296
  `len` function with, 297
  methods, 300–306
  as mutable, 294, 298–300
  negative indexes with, 296
  passing as arguments, 311–312
  processing, 308–309
  processing example, 314–317
  returning from functions, 312–314
  size, 310
  slicing, 297
  storing objects in, 356–358
  string, 295
  values, averaging, 310–311
  values, totaling, 310
  working with, 294, 317–320
Literals
  numeric, 46
  string, 37–38
Loan qualifier example programs, 141–142
Local variables. See also Functions; Variables
  creating, 91
  defined, 91
  errors, 91
  hidden, 92
  scope and, 91–93
  use example, 92–93
`log10()` function, 213
`log()` function, 213
Logic
  end of file detection, 244
  `if-elif-else` statement, 137
  input validation loops, 180
  mainline, 82
  running total calculation, 173
  `while` loop, 153
Logic errors, 32
Logical operators. See also Operators
  `and`, 138,139
  compound Boolean expressions using, 138
  defined, 138
  `not`, 138, 139–140
  numeric ranges with, 143
  `or`, 138,139–140
Long statements, breaking, 61–62
Loops
  condition-controlled, 152–161
  count-controlled, 152
  defined, 152
  in file processing, 242–249
  `for`, 161–172,245–246
  infinite, 159
  input validation, 179–184
  nested, 184–186
  pretest, 156–157
  recursion versus, 413
  sentinels and, 176–179
  validation, 210
  `while`, 152–161
`lower()` method, 286,287
Low-level languages, 16
`lstrip()` method, 286

M

Machine language, 13, 14, 15
Main memory
  defined, 4
  illustrated, 5
  programs in, 14–15
  as random-access memory (RAM), 5
Mainline logic, 82
`make_list` function, 358
Mammal class
  code, 388–389
  `__init__` method, 389
  `make_sound` method, 389, 391
  `show_species` method, 389
Math expressions
  assignment statements, 50–51
  defined, 50
  example, 50
  mixed-type, 59
  operands, 50
  using list elements in, 308–309
  value-returning functions in, 202
  variables, 50
`math` module
  contents, 211
  `e` variable, 213
  functions, 211–212, 213
  `pi` variable, 213
Math operators. See also Operators
  defined, 50
  precedence, 53–54
`max` function, 306–307
Memory
  flash, 6
  main, 4–5
  secondary storage, 5–6
  volatile, 5
Memory sticks, 6
Menu-driven programs, 218
`message` function
  calls, 401
  `main` function calling, 401
  string display, 400
Methods
  accessor, 355
  `append()`, 300–302
  call format, 284
  class, 336
  `close`, 231
  `convert`, 436
  defined, 230, 330
  `endwith()`, 288
  `find()`, 288
  `index()`, 301,302–304
  initializer, 336
  `__init__`, 336,345, 347,376,378
  `insert()`, 301,304
  `isalnum()`, 285
  `isalpha()`, 285
  `isdigit()`, 285
  `islower()`, 285
  `isspace()`, 285
  `isupper()`, 285

  - list, 300–306
  - `lower()`, 286,287
  - `lstrip()`, 286
  - modification, 286–287
  - mutator, 355
  - outside code interaction, 331
  - overriding, 388
  - private, 332
  - public, 332
  - `quit`, 432
  - `read`, 232,233
  - `readline`, 233–234,236, 240,241,244
  - readlines, 318
  - `remove()`, 301,305–306
  - `replace()`, 288,289
  - `reverse()`, 301,306
  - `rstrip`, 237,286
  - searching and replacing, 287–289
  - `self` parameter, 350
  - `sort()`, 301,304–305
  - `split()`, 320–321
  - `startswith()`, 288
  - string, 284–289
  - `strip()`, 286
  - `__str__`, 347–350
  - subclass, 388
  - superclass, 388
  - testing, 284–286
  - `upper()`, 286–287
  - write, 230
  - `writelines()`, 317
- Microprocessors, 3, 4
  - companies, 14
  - defined, 3
  - illustrated, 14
- `min` function, 306–307, 315
- Minimum field .widths
  - applying, 68
  - defined, 66
  - number display and, 67
  - specifying, 66–67
- Mixed-type expressions, 59
- Mnemonics, 16
- Modification methods. ***See also*** Methods
  - defined, 286
  - example use, 287
  - list of, 286
- Modularization, 214
- Modules
  - in code reuse, 214
  - defined, 192, 214
  - importing, 215, 343
  - `math`, 211–214
  - names, 215
  - `os`, 258
  - `random`, 193
  - storing classes in, 343–345
  - storing functions in, 214–218
  - `Tkinter`, 421–424
  - `tkMessageBox`, 430–433
- Multiple items
  - displaying with + operator, 63–64
  - displaying with `print` statement, 44
- Multiplication (*) operator, 53
- Music storage, 13
- Mutator methods, 355

N

- Negative indexes, 296
- Nested blocks. ***See also*** Blocks; Decision structures
  - defined, 120
  - illustrated, 121
- Nested decision structures
  - defined, 130
  - flowchart, 132
  - grade determination example, 135
  - identification, 133
  - multiple, 134–136
  - sequence structures in, 131
- Nested `if-else` statements
  - clause alignment, 133–134
  - example use, 141
  - long series of, 137
- Nested loops. ***See also*** Loops
  - defined, 184
  - example, 184–185
  - flowchart, 185
  - intermost loop, 185, 186
  - iteration total, 186
  - outer loop, 186
- Newline (\n) character
  - complications, 236
  - concatenating to strings, 235–236
  - defined, 62, 232
  - functions, 232
  - purpose inside files, 236
  - `readline` method and, 233
  - `rstrip` method and, 237
  - stripping from strings, 236–238
- `not in` operator
  - finding list items with, 297–298
  - testing strings with, 283
- `not` operator
  - defined, 138
  - truth table, 140
  - use example, 140
- Nouns
  - elimination, 361–363
  - identical meanings, 361–362
  - identifying, 360–361
  - list, refining, 361–364
  - non-essential item representation, 362
  - object-represented, 363
  - value-represented, 363–364
- Numbers
  - advanced storage, 12
  - data types, 45–46
  - factorial, calculating, 403–405
  - floating-point, 64
  - formatting, 64–69
  - integers, 45
  - negative, in indexes, 277, 280
  - nonnegative, factorial, 403
  - random, 192–199
  - ranges, changing, 143
  - reading with `input` function, 48–49
  - real, 45
  - storage, 9–11
- Numeric data
  - reading, 240–241
  - writing, 238–240
- Numeric literals, 46

O

- Object-oriented programming (OOP)
  - data hiding, 330
  - defined, 329
  - encapsulation, 330
  - premise, 330
- Objects. ***See also*** Methods
  - classes versus, 333
  - data attributes, 330, 340–343
  - defined, 330
  - elements, 330
  - everyday example, 331–332
  - reusability, 331
  - state, 347
  - storing in lists, 356–358

- Open function
  - call example, 229–230
  - defined, 229
  - general format, 229
- Operands, 50
- Operation systems, 6
- Operators
  - addition (+), 53, 307
  - `and`, 138–140
  - assignment, 175–176
  - defined, 18
  - division (/), 53
  - `in`, 283,297–298
  - logical, 138–144
  - math, 50
  - multiplication (*), 53
  - `not`, 138,140
  - `not in`, 283,298
  - `or`, 138–140
  - precedence, 53–54
  - relational, 116, 128
  - remainder (%), 53, 56
  - repetition (*), 292–293
  - string format, 64, 65
  - subtraction (–), 53
- `or` operator
  - defined, 138
  - short-circuit evaluation, 140
  - truth table, 139
  - use example, 139
- `os` module, 258
- Output
  - data, 62–69
  - defined, 6
  - devices, 6
  - displaying, 36–39
  - flowchart symbols, 34
  - program, 35, 36
- Output files
  - closing, 231
  - defined, 226
  - opening, 231

P

- Parameter lists, 98
- Parameters
  - changes, 99–101
  - defined, 93, 94
  - example, 94
  - passing by position to, 98
  - scope, 95
- Parentheses (()), 54
- Pascal, 17
- Pass by value, 101
- Passing arguments. *See* also Arguments
  - defined, 93
  - example, 96–97
  - multiple, 97–99
  - parameter variables, 94–95
  - by position, 98
  - by value, 101
- Passing lists, 311–312
- Pausing program execution, 90
- Percentages, calculating, 51–52
- Pixels, 12
- Polymorphism
  - behavior elements, 388
  - defined, 388
  - program design flexibility, 391
- Precedence, operator, 53–54
- Pretest loops
  - defined, 156
  - `while` loop as, 156–157
- Priming read, 180, 244
- `print` statement
  - example, 36
  - in list display, 295
  - in multiple item display, 44
  - newline suppression, 62
  - output display with, 36–39
  - string literals, 37–38
- Private methods, 332
- Problem domain
  - class responsibilities and, 365
  - defined, 360
  - description, 360
  - example, 360
  - noun identification, 360–361
  - noun list, refining, 361–364
- Problem solving with recursion, 402–406
  - defined, 402
  - number factorial calculation, 403–405
  - overhead, 402
  - repetitive, 402
- Procedural programming, 329
- Procedures
  - creation, 330
  - operation, 329
- Process, program, 35, 36
- Processing symbols, 34
- Program design, 31–35
  - flowcharts, 34–35, 85–86
  - for function use, 85–90
  - hierarchy charts, 87
  - polymorphism and, 391
  - process, 32
  - pseudocode, 34
  - steps determination, 33–34
  - top-down, 86–87
  - with `while` loops, 157–159
- Program development cycle
  - defined, 31
  - elements, 32
  - illustrated, 31
- Programmers
  - customer interview, 33
  - defined, 1
  - task breakdown, 33–34
- Programming
  - object-oriented, 329, 330–332
  - procedural, 329–330
- Programming languages. *See* also Python
  - Ada, 17
  - assembly, 15–16
  - BASIC, 17
  - C#, 17
  - C/C++, 17
  - COBOL, 17
  - compilers/interpreters, 18–20
  - FORTRAN, 17
  - high-level, 16–17
  - Java, 17
  - JavaScript, 17
  - key words/reserved words, 18
  - low-level, 16
  - operators, 18
  - Pascal, 17
  - Ruby, 17
  - source code, 20
  - statements, 18
  - syntax, 18
  - Visual Basic, 17
- Programs. *See* also Python programs
  - defined, 1
  - event-driven, 421
  - exceptions, 262–268
  - execution, pausing, 90
  - functioning of, 13–20
  - GUI, 421
  - image editing, 2
  - input, 35–36
  - line numbers, 37
  - in main memory, 14–15
  - mainline logic, 82
  - menu-driven, 218
  - output, 35, 36
  - process, 35, 36

    storage, 14
    task, steps, 33–34
    task, understanding, 32–33
    testing, 32
    three-step process, 36
    utility, 7
    word processing, 2
Pseudocode, 34
Public methods. *See* also Methods
    attribute access, 355
    defined, 332
Python 2.5, 457
Python
    defined, 17
    directory,. adding to Path variable, 458
    documentation, 457
    IDLE, 23–24
    installing, 21, 457
    interactive mode, 21–22
    key words, 18
    operators, 18
    quitting, 22
    script mode, 21, 22–23
    Shell window, 460, 461, 466
    using, 20–24
    Windows installer, 457
Python interpreter, 20, 21, 457
    interactive mode, 459
    multiline statement execution, 461
    statements executed by, 460
    wait prompt (>>>), 460
Python programs
    defined, 21
    naming, 23
    running, 23, 465–466
    saving, 23, 464
    writing, 22

Q
Quit button, 432-433, 434
quit method, 432
Quote marks, for string literals, 37–38

R
`radians()` function, 213
Radio buttons, 444
`Radiobutton` widgets
    callback functions with, 447–448
    clicking, 445
    selection, 444–445
`randint` function
    call example, 193
    defined, 193
    integer value return, 197
    program example, 194–195
    in random number generation, 335
    values returned by, 194
Random access files. *See* Direct access files
random function, 199
random module, 193
Random numbers
    displaying, 195
    generating, 192–199
    library functions, 193
    to represent other values, 197–198
    uses, 192–193
    using, 196–197
Random-access memory (RAM), 5
randrange function
    defined, 198
    example uses, 198–199
range function
    arguments passed to, 165
    defined, 164
    with for loop, 164–166
    in list generation, 295
raw— input function, 48, 49
read method, 232
Read position
    advanced to end of file, 235
    advanced to next line, 234
    defined, 234
    initial, 234
Reading
    data from files, 227, 232–235
    files with loops, 243–245
    input from keyboard, 48–50
    numbers, 48–49
    numeric data, 240–241
    records, 251
    strings, 49
`readline` method, 233–234, 236,240,241
    empty string return, 244
`readlines()` method, 318
Real numbers, 45
Records
    adding, 253–255
    copying, 260
    defined, 249, 250
    deleting, 260–261
    displaying, 253–255
    modifying, 257–259
    processing, 249–262
    reading, 253
    searching for, 255–257
    writing, 250
Recursion
    depth of, 402
    direct, 406
    indirect, 406
    introduction to, 399–402
    in number factorial calculation, 403–405
    problem solving with, 402–406
Recursive algorithms
    designing, 403
    Fibonacci series, 407–409
    GCD, 409–410
    looping versus, 413
    summing list elements range, 406–407
    Towers of Hanoi, 410–413
Recursive case, 403
Recursive functions
    controlling, 400,402
    defined, 399
    Fibonacci series calculation, 408
    functioning of, 403
    program example, 399–400
Relational operators
    Boolean expressions with, 116
    defined, 116
    list of, 116
    in string comparisons, 128
Remainder (%) operator. *See* also Operators
    defined, 56
    precedence, 53
    using, 56
remove() method, 301
rename function, 258
Repetition operator (")
    defined, 292
    in list creation, 295
    program example, 292–293
    use example, 292
Repetition structures. *See* also Loops
    defined, 151
    example, 151–152
    introduction to, 151–152

`replace()` method, 288,289
Reserved words, 18
`return` statement. *See* also Value-returning functions
  defined, 200
  using, 202
  values, 211
`Reverse()` method
  defined, 301
  use example, 306
Review questions
  classes and OOP programming, 367–370
  computers and programming introduction, 24–27
  decision structures, 145–148
  files and exceptions, 268–271
  functions, 107–110
  GUI programming, 451–453
  inheritance, 394–396
  input, processing, and output, 69–73
  repetition structures, 186–188
  strings and lists, 323–326
  value-returning functions and modules, 218–220
Rounding, dollar amounts, 58
`rstrip()` method, 237,286
Ruby programming language, 17
Running totals
  accumulator, 173
  calculating, 173–176
  defined, 173
  example, 174
  logic, 173

S

Samples, 13
Scope. *See* also Variables
  defined, 91
  local variable, 91–93
  parameter, 95
Script mode
  defined, 21
  using, 22–23
Scripts, 21
Searching
  methods, 287–289
  for records, 255–257
Secondary storage
  defined, 5
  types of, 5–6
Selection structures. *See* Decision structures
`self` parameter, 350
Sentinels
  defined, 176, 177
  using, 177–179
  values, 177
Separators, 321
Sequences
  arrays, 295
  with decision structure, 130
  defined, 113, 273
  items in, 273
  lists, 294–322
  strings, 273–294
  types of, 273
Sequential access files. *See* also Files
  defined, 227–228
  record modification, 257
  working with, 228
`ServiceQuote` class, 366–367
Short-circuit evaluation, 140
`showinfo` function, 430, 432
`sin()` function, 213
Single alternative decision structure, 114
Single-quotes, 38
Slices
  defined, 279
  examples, 280
  expression format for, 279
  list, 297
Software. *See* also Programs
  application, 7
  defined, 1
  developers. *See* Programmers
  requirement, 33
  system, 6–7
Software development tools, 7
`sort()` method
  defined, 301
  use examples, 304–305
Source code
  defined, 20
  functions and, 79
  writing, 32
Specialization, 373–374
`split()` method
  calling, 321
  defined, 320
  program examples, 320–322
Splitting strings
  program examples, 320–322
  with `split` method, 320–322
Spreadsheets, as files, 225
`sqrt` function, 212, 213
Standard library functions, 192
`startswith()` method, 288
Statements
  converting math formulas to, 56–58
  defined, 18
  `del`, 306
  `for`, 152
  `if`, 113–121
  `if-elif-else`, 136–138
  `if-else`, 121–125
  `if-then`, 119–120
  `import`, 192,193
  line continuation character ('), 61
  long, breaking, 61
  `return`, 200,202
  saving, 22
  `try/except`, 264–265
  `while`, 152
Step values, 165
`str` data type, 46
`str` function, 238–239
String concatenation
  + operator, 63
  defined, 63–64
  uses, 64
String format operator
  defined, 64
  formatting one value with, 65
  formatting several values with, 66
  use example, 65
String literals
  apostrophes, 38
  defined, 37
  examples, 37, 38
  quote marks, 37–38
  single-quotes, 38
Strings
  character access, 274–278
  character copies, getting, 277
  characters, extracting, 281–283
  characters, validating, 289–292
  comparing, 126–130
  concatenating newlines to, 235–236
  defined, 37
  formatting, 67–69
  as immutable, 278–279

  indexing, 276–278
  iterating over with for loop, 274–276
  list, 295
  methods, 273, 284–289
  reading, 274
  reading with raw—input function, 49
  readinglstripping newline from, 236–238
  returning, 208–209
  slicing, 279–280
  splitting, 320–322
  storing with str data type, 46–42
  testing, 283
  variable assignment, 46–47
  working with, 273–294
  writing, 274
StringVar object
  defined, 436
  with Label widget, 436–440
  reference storage, 440
  set method, 440
Strip() method, 286
__str__ method, 347–350
  automatic calling, 350
  defined, 348
  program example, 348–350
Structure charts. *See* Hierarchy charts
Subclasses
  defined, 374
  inheriting from classes, 377
  method overriding, 388
  methods, 388
Substrings
  defined, 279
  examples, 280
  expression format for, 279
Subtraction (–) operator, 53
sum—range function, 406–407
Superclasses
  defined, 374
  methods, 388
SUV class
  code, 380
  defined, 375
  get—pass_cap method, 381
  __init__ method, 380
  setpass— cap method, 381
  in UML diagram, 383
Syntax
  defined, 18
  rules, 20
Syntax errors
  correcting, 32
  defined, 20
System software, 6–7

T

tan() function, 213
Target variables
  defined, 163
  inside loops, 166–168
  purpose, 166
Temporary files
  copying records to, 257,260
  creating, 257
  renaming, 257, 260
Terminal symbols, 34
Testing
  functions and, 79
  string methods, 284–286
  strings, 283
Testing programs, 32
Text
  button, 430
  displaying in window, 424–427
  editor, 23
  files, 227
Tk library, 421
Tkinter module. See also GUI programs
  Checkbutton class, 448
  defined, 421
  in GUI program creation, 421–424
  IDLE use, 422
  importing, 423
  IntVar, 445,448
  mainloop function, 424,425
  program examples, 422–424
  program use, 422
  Radiobutton class, 444–448
  StringVar class, 436
  widgets, 422
TkMessageBox module, 430–433
Top-down design. *See* also Program design
  defined, 86
  process, 86–87
Totals
  list value, 310
  running, 173–176
Towers of Hanoi
  algorithm summary, 411–412
  defined, 410
  game play, 410–411
  overall solution, 411
  peg-moving steps, 411
  pegs and discs, 410
  program example, 412–413
Tracebacks, 263
Triple-quotes, 38
Truck class
  code, 379
  defined, 375
  get_drive method, 380
  __init__ method, 380
  set_drive method, 380
  in UML diagram, 383
True variable, 144
Truth tables
  and operator, 139
  not operator, 140
  or operator, 139
try/except statements
  defined, 264
  event sequence, 266
  except clauses, 265, 268
  execution, 265
  general format, 264
  try block, 264–265
  use example, 265–266

U

UML diagrams
  Car class, 366
  CellPhone class, 359
  Coin class, 359
  Customer class, 365
  defined, 358
  examples, 359
  general layout for classes, 359
  inheritance in, 382–383
  ServiceQuote class, 367
Unicode, 12
Unified Modeling Language. *See* UML diagrams
Uniform function, 199
upper() method, 286,287
USB drives, 5–6

User interfaces
  command line, 419–420
  defined, 419
  graphical (GUI), 419–450
Users, 49
Utility programs, 7

V

Validation code, 210
Validation loops, 210
`ValueError` exception, 267, 268
Value-returning functions
  benefits, 202
  defined, 191
  example, 200
  in mathematical expressions, 202
  modularizing with, 205–208
  parts of, 200
  `return` statement, 200, 202
  returning Boolean values, 209–210
  returning multiple values, 212
  returning strings, 208–209
  as simple functions, 191
  using, 202–204
  values, 191, 200
  writing, 200–211
Values
  Boolean, returning, 209–210
  different per execution, 176
  long list, processing, 177
  multiple, returning, 211
  passing by, 101
  random number to represent, 197–198
  `return` statement, 211
  returned by `randint`, 194
  sentinel, 177
  step, 165
  value-returning functions, 191, 200
Variables
  accumulator, 173, 266
  Boolean, 144–145
  camelCase, 43–44
  creating inside functions, 103
  creating outside functions, 103
  creating with assignment statements, 40–42
  defined, 40
  errors, 42
  examples, 40, 41, 42
  first letter, 43
  global, 103–105
  key words and, 43
  local, 91–93
  in math expressions, 50
  names, sample, 43, 44
  names, selecting, 43
  naming rules, 43–44
  parameter, 93–94
  `Print` statement and, 44
  program use, 40
  reassignment, 45
  scope, 91
  string assignment, 46–47
  target, 163
  upper/lowercase letters, 42
  use warning, 42
  value representation, 40
Verbs, in function names, 80
Visual Basic, 17

W

`while` loops, 253
  beginning of, 154
  condition tested by, 153, 155
  defined, 152
  flowchart, 153, 156
  function calls, 160–161
  infinite, 159
  logic, 153
  parts, 153
  as pretest loop, 156–157
  program design with, 157–159
`while` statement, in condition-controlled loops, 152
Widgets
  arrangement of, 430
  `Button`, 430–433
  defined, 422
  `Entry`, 433–436
  Frames, 428
  `Label`, 424–427
  organizing with `Frames`, 427–430, 441
  `quit` method, 432
  `Radiobutton`, 444–448
  root, 424, 432
  `Tkinter`, 422
Windows Explorer, 228
Windows Vista, 458
Windows XP, 458
`Writelines()` method, 317
Writing
  comments, 40
  data to files, 226, 230–232
  dual alternative decision structures, 122
  function definitions, 80–81
  input validation loops, 181–184
  numeric data, 238–240
  records, 250
  source code, 32
  value-returning functions, 200–201

Z

`ZeroDivisionError` exception, 268

PLEASE SEE REVERSE SIDE FOR INSTRUCTIONS TO OPEN

ING OUT WITH
PYTHON
TONY GADDIS
ISBN: 0321537114
ADDISON-WESLEY
HIS CD INCLUDES:
Python® 2.5.1 Installer
Microsoft Window
IDLE)
including
ource code
Appendix D:
Answers to Checkpo
Open Start_Here.html to access the resources
on this CD.
please check our support page at
http://www.aw.com/cssupport for updates
to these resources.
STUDENT
RESOURCE
DISK

Introductory Programming / Python®

Gaddis Books—Understanding from the Start!

Starting Out with Python introduces students to the basics of programming and prepares them to go on to more complicated languages. With the knowledge acquired by using Python, students gain confidence in their skills and acquire the logic necessary for developing high-quality programs. This book discusses control structures, functions, lists, and file I/O before introducing classes. As with all Gaddis texts, clear and easy-to-read code listings, concise and practical real-world examples, detail-oriented explanations, and an abundance of exercises appear in every chapter.

This book is ideal for a one-semester introductory programming course for students with limited programming experience.

COMPATIBLE WITH OTHER BOOKS IN THE GADDIS SERIES

Tony Gaddis's best-selling *Starting Out With* series provides accessible, detailed presentations of programming concepts using an approach that will increase confidence and competence in novice programmers.

The *Starting Out With* series includes books that meet most course and teaching styles. Titles include *Starting Out with Alice, Python, Visual Basic®, C++, and Java™*.

Visit www.aw.com/gaddisbooks for more information.

KEY FEATURES

- **Control structures, functions, lists, and file I/O** are covered before classes are introduced.
- **A clear and student-friendly writing style** simplifies programming processes for beginning programmers, with two to three stepped-out explanations following each major concept.
- **A variety of exercises** in each chapter encourages students to put concepts to work as they are learned. Source code is provided so that students can run the programs themselves.
- **In the Spotlight** case studies walk students through the design of a problem by showing all the important steps.
- **Checkpoints** check students' understanding of important lessons at key places in each chapter. Other student-friendly features include Concept, Note, Tip, and Warning boxes.
- **A student resource CD-ROM** packaged with each new text includes Python, the IDLE development environment, source code from the example problems, and answers to all Checkpoint questions.

PEARSON
Addison Wesley

Visit www.aw.com/computing for more information about Addison-Wesley computing books.
To order any of our products, contact our customer service department at
(800) 824-7799 or (201) 767-5021 outside of the U.S., or visit your campus bookstore.

Addison-Wesley Computing
Leading Authors • Quality Products

ISBN-13: 978-0-321-53711-9
ISBN-10: 0-321-53711-4
EAN
9 780321 537119
90000